Vamos falar sobre as diferenças do comando `switch` no Java 8 e no Java 17. Muita coisa mudou.

Switch no Java 8

No Java 8, o `switch` era bastante simples e limitado em termos de funcionalidades. Ele suportava **apenas tipos primitivos como** `int`, `char`, `short`, `byte`, e seus equivalentes `Integer`, `Character`, `Short`, `Byte`, além de `String` e enums. Vamos dar uma olhada em como isso funciona com um exemplo.

```
public class SwitchExampleJava8 {
    public static void main(String[] args) {
        int day = 2;
        String dayString;

        switch (day) {
            case 1:
                dayString = "Domingo";
                break;
            case 2:
                dayString = "Segunda-feira";
                break;
            case 3:
                dayString = "Terça-feira";
                break;
            case 4:
                dayString = "Quarta-feira";
                break;
            case 5:
                dayString = "Quinta-feira";
                break;
            case 6:
                dayString = "Sexta-feira";
                break;
            case 7:
                dayString = "Sábado";
                break;
            default:
                dayString = "Dia inválido";
                break;
        }

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

Características do `switch` no Java 8:

1. **Uso de `break`**: Necessário para evitar o *fall-through* (quando um case executa o próximo caso se não houver `break`).

2. **Tipos suportados**: Apenas tipos primitivos, `String` e enums, entretanto os tipos primitivos `boolean`, `long` e `float` não são aceitos.
3. **Sintaxe**: Estritamente estruturada, com múltiplas linhas de código.
4. O bloco `default` é opcional.
5. Você não precisa de `{}` no bloco de comando

Switch no Java 17

No Java 17, o `switch` recebeu grandes melhorias. Ele agora é mais expressivo, flexível e inclui suporte para *pattern matching* e *switch expressions*. Vamos ver essas mudanças com exemplos.

1. Switch Expressions

Java 17 introduziu as switch expressions, que permitem retornar valores diretamente do `switch`.

```
public class SwitchExampleJava17 {
    public static void main(String[] args) {
        int day = 2;
        String dayString = switch (day) {
            case 1 -> "Domingo";
            case 2 -> "Segunda-feira";
            case 3 -> "Terça-feira";
            case 4 -> "Quarta-feira";
            case 5 -> "Quinta-feira";
            case 6 -> "Sexta-feira";
            case 7 -> "Sábado";
            default -> "Dia inválido";
        };

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

`switch` pode ser tratado agora como uma expressão, é necessário o ponto e vírgula após o bloco `{}`

Perceba também que não é mais necessário o `break` para evitar o "*fall-through*".

2. Setas (`->`) e `yield`

No Java 17, usamos setas (`->`) em vez de `case` e `break`. Para casos mais complexos, onde múltiplas linhas de código são necessárias, usamos `yield`.

```
public class SwitchExampleJava17Complex {
    public static void main(String[] args) {
        int day = 2;
        String dayString = switch (day) {
            case 1 -> "Domingo";
            case 2 -> {
                System.out.println("Executando caso da Segunda-feira");
                yield "Segunda-feira";
            }
            case 3 -> "Terça-feira";
            case 4 -> "Quarta-feira";
            case 5 -> "Quinta-feira";
            case 6 -> "Sexta-feira";
            case 7 -> "Sábado";
            default -> "Dia inválido";
        };

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

Outro exemplo do uso do `yield`:

```
public class GreetingExample {
    public static void main(String[] args) {
        GreetingExample example = new GreetingExample();
        example.greet(1, -1);
    }

    public void greet(int a, int b) {
        String greeting = switch (a) {
            case 0 -> "Good morning";
            case 1,2,3 -> {
                if (b > 0) yield "Good morning";
                else yield "Good afternoon";
            }
            case 4 -> "Good evening";
            default -> "Hello";
        };
        System.out.println(greeting);
    }
}
```

Você pode usar o `yield` em uma única linha de comando, mas é obrigatório uso de `{}`.

3. Pattern Matching

Pattern matching em `switch` é uma grande novidade, permitindo fazer match de objetos com seus tipos e atributos. Nas versões anteriores do Java, a expressão do seletor era limitada a apenas alguns tipos. Entretanto, com padrões de tipo, a expressão do seletor de chave pode ser de qualquer tipo.Atualmente, está em prévia, mas dá para ver um exemplo básico:

```
public class SwitchPatternMatching {
    public static void main(String[] args) {
        Object obj = "Olá, Java 17!";

        switch (obj) {
            case String s -> System.out.println("String com valor: " + s);
            case Integer i -> System.out.println("Inteiro com valor: " + i);
            default -> System.out.println("Tipo desconhecido");
        }
    }
}
```

Você deve explicitamente habilitar as features do Java que estão no modo *preview*:

```
java --enable-preview --source 17 SwitchPatternMatching.java
```

4. Mais exemplos:

A expressão switch pode retornar diferentes tipos de valores:

```
public class Greeting {

    public void greet(int a) {
        var printOut = switch (a) {
            case 0 -> "Good morning"; // String
            case 1 -> 7; // int
            case 2,3,4 -> true; // boolean
            default -> 3.14; // double
        };
        System.out.println(printOut);
    }

    public static void main(String[] args) {
        Greeting greeting = new Greeting();
        greeting.greet(0); // Teste com 0
        greeting.greet(1); // Teste com 1
        greeting.greet(2); // Teste com 2
        greeting.greet(3); // Teste com default case
    }
}
```

Uma expressão switch obrigatoriamente deve lidar com todos os casos possíveis. O código abaixo NÃO vai compilar:

```
public class Greeting {

    public void greet(int a) {
        var printOut = switch (a) {
            case 0 -> "Good morning";
            case 1 -> "Good afternoon";
            case 2 -> "Good evening";
        };
        System.out.println(printOut);
    }

    public static void main(String[] args) {
        Greeting greeting = new Greeting();
        greeting.greet(0); // Teste com 0
        greeting.greet(1); // Teste com 1
        greeting.greet(2); // Teste com 2
    }
}
```

`int a` possue infinitos valores, logo precisamos sempre adicionar uma cláusula de `default` . O código corrigido:

```
public class Greeting {

    public void greet(int a) {
        var printOut = switch (a) {
            case 0 -> "Good morning";
            case 1 -> "Good afternoon";
            case 2 -> "Good evening";
            default -> "Invalid option"; // Adicionando um caso default para
segurança
        };
        System.out.println(printOut);
    }

    public static void main(String[] args) {
        Greeting greeting = new Greeting();
        greeting.greet(0); // Teste com 0
        greeting.greet(1); // Teste com 1
        greeting.greet(2); // Teste com 2
        greeting.greet(3); // Teste com um caso inválido
    }
}
```

No caso de variáveis do tipo `enum` não precisamos usar default, se endereçarmos **todos** os casos possíveis pois `enum` possui um número limitado de valores:

```
public class Direction {

    enum Compass {
        NORTH, SOUTH, EAST, WEST
    }

    String getDirection(Compass value) {
        return switch (value) {
            case NORTH -> "Up";
            case SOUTH -> "Down";
            case EAST -> "Right";
            case WEST -> "Left";
        };
    }

    public static void main(String[] args) {
        Direction direction = new Direction();
        System.out.println(direction.getDirection(Compass.SOUTH)); // Deve
imprimir "Down"
        System.out.println(direction.getDirection(Compass.NORTH)); // Deve
imprimir "Up"
        System.out.println(direction.getDirection(Compass.EAST));  // Deve
imprimir "Right"
        System.out.println(direction.getDirection(Compass.WEST));  // Deve
imprimir "Left"
    }
}
```

Em Java, os rótulos dos casos de um `switch` devem ser constantes. No caso do `switch` expression, todas as possíveis opções (`case`) precisam ser constantes em tempo de compilação. O código abaixo só **compila pois a variável `b` é do tipo `final`** (constante).

```
public class SwitchExampleSelectorExpression {
    public static void main(String[] args) {
        final String b = "B";
        switch (args[0]) {
            case "A" -> System.out.println("Parâmetro é A");
            case b -> System.out.println("Parâmetro é B");
            default -> System.out.println("Parâmetro é desconhecido");
        }
    }
}
```