Aula sobre
Switch Case em Java

1. Parte teórica – Switch em Java

1.1 O que é o `switch` e quando usar

Definição:

`switch` é uma estrutura de controle que permite **desviar o fluxo** do programa com base no valor de uma expressão, comparando-o contra vários **valores constantes** (`case`).

É uma alternativa a uma sequência longa de `if/else if` quando:

- Você está comparando **o mesmo valor** contra **várias constantes** (números, `String` , `enum` etc).
- As condições são **exatas** (igualdade), não intervalos (`>` , `<` , etc).
- Você quer código **mais legível** e organizado que um monte de `if/else` .

Quando usar:

- Menus de opções.
- Mapeamento de **códigos** (status numérico → texto, código de erro → mensagem).
- Lógica baseada em **estado** (status de pedido, tipo de usuário, etapa de workflow).
- Tratamento de **comandos** (ex: "CREATE", "UPDATE", "DELETE").

1.2 Como funciona internamente (visão simplificada)

Por baixo dos panos, para tipos numéricos e `enum` , o compilador muitas vezes gera instruções tipo:

- `tableswitch` → quando os valores são consecutivos ou quase.
- `lookupswitch` → quando são esparsos.

Na prática:

- Em vez de vários `if` s, o JVM faz uma espécie de **"tabela de saltos"** (jump table).
- Isso pode ser **mais eficiente** que vários `if/else` , especialmente com muitos casos.

Mas em alto nível, para o aluno:

> O switch pega o valor da expressão, compara com cada `case` , e quando encontra um caso compatível, executa aquele bloco.

1.3 Diferença entre `if/else` e `switch`

Use `if/else` quando:

- As condições são **intervalos**: `x > 10 && x < 20`.
- Há combinações complexas: `if (idade > 18 && renda > 5000)`.
- Você precisa de qualquer lógica booleana, não apenas igualdade.

Use `switch` quando:

- Você tem **um valor único** sendo comparado com **vários valores fixos**.
- Exemplo: `tipoUsuario = "ADMIN" | "CLIENTE" | "GERENTE"`.

Leitura:

- `if/else` pode ficar verboso com muitos casos.
- `switch` fica mais "limpo" quando há muitas alternativas bem definidas.

1.4 Regras, sintaxe e limitações (switch tradicional)

Sintaxe básica (tradicional):

java Copiar código

```java
switch (expressao) {
    case VALOR1:
        // código
        break;
    case VALOR2:
        // código
        break;
    default:
        // código se nenhum caso combinar
}
```

Pontos importantes:

1. A expressão deve ser de um tipo suportado:
   - Tipos primitivos: `byte`, `short`, `char`, `int`
   - `enum`
   - `String` (desde Java 7)
   - Wrappers correspondentes (`Byte`, `Short`, `Character`, `Integer`)
2. **Não pode**: `long`, `float`, `double`, `boolean`, classes arbitrárias.
3. Os valores de `case` devem ser **constantes em tempo de compilação**:
   - `final int X = 10; case X:`
   - Literal: `case 1:`, `case "ADMIN":`

4. `break` é fundamental:
   - Sem `break`, acontece o **fall-through**: o fluxo continua para os próximos `case`.
5. `default`:
   - Opcional, mas recomendado.
   - Executado quando nenhum `case` combina.

1.5 Switch tradicional vs. Switch Expressions (Java 14+)

A partir de Java 14 (preview antes, definitivo depois), temos:

- **Switch Statement** (tradicional) – não retorna valor.
- **Switch Expression** – retorna um valor.

Exemplo tradicional:

java Copiar código

```java
int dia = 3;
String nomeDia;

switch (dia) {
    case 1: nomeDia = "Domingo"; break;
    case 2: nomeDia = "Segunda"; break;
    case 3: nomeDia = "Terça"; break;
    default: nomeDia = "Desconhecido";
}
```

Exemplo **switch expression**:

java Copiar código

```java
int dia = 3;

String nomeDia = switch (dia) {
    case 1 -> "Domingo";
    case 2 -> "Segunda";
    case 3 -> "Terça";
    default -> "Desconhecido";
};
```

Diferenças importantes:

- `->` elimina o problema do `break` (sem fall-through por padrão).

- Switch expression **sempre retorna** um valor.
- Pode usar `yield` dentro de blocos `{}` quando precisa de lógica mais extensa:

java Copiar código

```
String descricao = switch (status) {
    case PENDENTE -> "Aguardando pagamento";
    case PAGO -> "Pago";
    case CANCELADO -> {
        logCancelamento();
        yield "Cancelado";
    }
};
```

---

2. Exemplos práticos (8+ exemplos completos)

Cada exemplo: contexto → código → casos de uso → vantagens → cuidados → passo a passo.

Exemplo 1 – Switch simples com `int` (dia da semana)

Contexto do problema:
Você tem um sistema simples que recebe um número de 1 a 7 e precisa exibir o nome do dia da semana. Esse tipo de lógica aparece em scripts, agendamentos, relatórios.

Código completo:

java Copiar código

```
public class Exemplo01SwitchInt {

    public static void main(String[] args) {
        int dia = 3;

        String nomeDia;
        switch (dia) {
            case 1:
                nomeDia = "Domingo";
                break;
            case 2:
```

```
                nomeDia = "Segunda";
                break;
            case 3:
                nomeDia = "Terça";
                break;
            case 4:
                nomeDia = "Quarta";
                break;
            case 5:
                nomeDia = "Quinta";
                break;
            case 6:
                nomeDia = "Sexta";
                break;
            case 7:
                nomeDia = "Sábado";
                break;
            default:
                nomeDia = "Dia inválido";
        }

        System.out.println("Dia " + dia + " é: " + nomeDia);
    }
}
```

Casos de uso reais:

- Geração de relatórios por dia da semana.
- Regras de desconto diferentes por dia.
- Agendamentos e rotinas de sistemas batch.

Vantagens:

- Código mais legível que vários `if (dia == 1) ...` .
- Fácil de enxergar todos os casos possíveis.

Possíveis problemas/cuidados:

- Se você esquecer um `break` , pode cair em *fall-through*.
- Entrada inválida precisa ser tratada (`default`).

Passo a passo:

1. Declara `dia = 3` .
2. O `switch (dia)` compara `dia` com cada `case` .
3. Como `dia` é 3, entra no `case 3` .

4. Atribui `"Terça"` à variável `nomeDia`.
5. O `break` sai do `switch`.
6. Imprime o resultado.

Exemplo 2 – Switch com `String` (comandos de operação)

Contexto do problema:

Um sistema de console recebe comandos como `"CRIAR"`, `"ATUALIZAR"`, `"DELETAR"`. Você quer centralizar a lógica de roteamento.

Código completo:

java Copiar código

```java
public class Exemplo02SwitchString {

    public static void main(String[] args) {
        String comando = "CRIAR";

        switch (comando) {
            case "CRIAR":
                System.out.println("Executando lógica de criação...");
                // chamar serviço de criação
                break;
            case "ATUALIZAR":
                System.out.println("Executando lógica de atualização...");
                // chamar serviço de atualização
                break;
            case "DELETAR":
                System.out.println("Executando lógica de exclusão...");
                // chamar serviço de exclusão
                break;
            default:
                System.out.println("Comando inválido: " + comando);
        }
    }
}
```

Casos de uso reais:

- CLI (Command-line Interface).
- Bots simples.

- Endpoints que recebem parâmetros simples de ação.

Vantagens:

- Código muito legível: cada comando mapeado para um bloco.
- Fácil adicionar novos comandos.

Cuidados:

- Sensibilidade a maiúsculas/minúsculas (`"CRIAR" ≠ "criar"`).
- Em sistemas maiores, talvez valha migrar para Strategy.

Passo a passo:

1. `comando = "CRIAR"` .
2. `switch` compara com `"CRIAR"` , `"ATUALIZAR"` , `"DELETAR"` .
3. Bate no `case "CRIAR"` → executa o bloco.
4. `break` encerra o `switch` .
5. Se nenhum case bater, exibe mensagem de comando inválido.

Exemplo 3 – Switch com `enum` (status de pedido)

Contexto do problema:

Você tem um e-commerce com pedidos em estados: `NOVO` , `PAGO` , `ENVIADO` , `ENTREGUE` , `CANCELADO` . A mensagem para o cliente muda conforme o status.

Código completo:

java Copiar código

```java
enum StatusPedido {
    NOVO,
    PAGO,
    ENVIADO,
    ENTREGUE,
    CANCELADO
}

public class Exemplo03SwitchEnum {

    public static void main(String[] args) {
        StatusPedido status = StatusPedido.ENVIADO;

        String mensagem;
```

```
        switch (status) {
            case NOVO:
                mensagem = "Seu pedido foi recebido e está em análise.";
                break;
            case PAGO:
                mensagem = "Pagamento confirmado! Em breve seu pedido será envi
                break;
            case ENVIADO:
                mensagem = "Seu pedido foi enviado. Acompanhe o rastreio.";
                break;
            case ENTREGUE:
                mensagem = "Pedido entregue! Obrigado pela preferência.";
                break;
            case CANCELADO:
                mensagem = "Seu pedido foi cancelado. Entre em contato se tiver
                break;
            default:
                mensagem = "Status desconhecido.";
        }

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

Casos de uso reais:

- Status de pedidos, boletos, assinaturas.
- Workflow de atendimento (ABERTO, EM_ANDAMENTO, FECHADO).
- Processamento de Jobs (PENDENTE, PROCESSANDO, ERRO, FINALIZADO).

Vantagens:

- `enum` evita "string solta" e digitação errada.
- Switch fica expressivo e seguro em termos de tipo.

Cuidados:

- Se adicionar um novo valor no `enum` e esquecer de tratar no `switch`, pode cair no `default`.
- Em switch expression (mais moderno), o compilador força tratamento de todos os valores.

Passo a passo:

1. Define o `enum StatusPedido`.

2. Atribui `ENVIADO` ao `status`.
3. `switch (status)` avalia qual `case` bate.
4. Atribui a mensagem correspondente.
5. Imprime para o cliente.

Exemplo 4 – Menu de opções com `switch` (console)

Contexto do problema:

Sistema de console que apresenta um menu e executa a opção escolhida pelo usuário.

Código completo:

java Copiar código

```java
import java.util.Scanner;

public class Exemplo04MenuOpcoes {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("==== MENU BANCO ====");
        System.out.println("1 - Consultar saldo");
        System.out.println("2 - Depositar");
        System.out.println("3 - Sacar");
        System.out.println("0 - Sair");
        System.out.print("Escolha uma opção: ");

        int opcao = scanner.nextInt();

        switch (opcao) {
            case 1:
                System.out.println("Consultando saldo...");
                break;
            case 2:
                System.out.println("Realizando depósito...");
                break;
            case 3:
                System.out.println("Realizando saque...");
                break;
            case 0:
                System.out.println("Saindo do sistema...");
```

```
                break;
            default:
                System.out.println("Opção inválida!");
        }

        scanner.close();
    }
}
```

Casos de uso reais:

- Ferramentas internas de manutenção.
- Scripts administrativos.
- Sistemas legados de terminal.

Vantagens:

- Estrutura natural para menus.
- Fácil de entender para iniciantes.

Cuidados:

- Validar entrada (usuário pode digitar letra ao invés de número → `InputMismatchException`).
- Em sistemas reais, separar lógica em métodos/service.

Passo a passo:

1. Exibe menu.
2. Lê a opção via `Scanner`.
3. `switch` decide qual bloco executar.
4. Cada `case` representa uma ação do menu.

Exemplo 5 – Validação de entrada do usuário (conceito de nota)

Contexto do problema:
Aluno digita uma letra (`A`, `B`, `C`, `D`, `E`) representando conceito de nota. O sistema precisa exibir uma mensagem.

Código completo:

java Copiar código

```
import java.util.Scanner;
```

```java
public class Exemplo05ValidacaoEntrada {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Informe o conceito (A, B, C, D, E): ");
        String input = scanner.nextLine().trim().toUpperCase();

        if (input.length() != 1) {
            System.out.println("Entrada inválida. Digite apenas uma letra.");
            scanner.close();
            return;
        }

        char conceito = input.charAt(0);

        switch (conceito) {
            case 'A':
                System.out.println("Excelente!");
                break;
            case 'B':
                System.out.println("Muito bom!");
                break;
            case 'C':
                System.out.println("Bom.");
                break;
            case 'D':
                System.out.println("Precisa melhorar.");
                break;
            case 'E':
                System.out.println("Reprovado.");
                break;
            default:
                System.out.println("Conceito inválido!");
        }

        scanner.close();
    }
}
```

Casos de uso reais:

- Sistemas acadêmicos (conceito de provas).
- Classificação de satisfação (A–E, S/N, Y/N).

- Entrada de códigos simples via teclado.

Vantagens:

- Simples de implementar.
- Switch com `char` fica muito limpo.

Cuidados:

- Sempre normalizar (`toUpperCase` , `trim`).
- Validar tamanho da string antes de acessar `charAt(0)` .

Passo a passo:

1. Lê a string do usuário.
2. Normaliza para maiúscula e remove espaços.
3. Verifica se tem exatamente 1 caractere.
4. Converte para `char` .
5. `switch` decide mensagem com base em `conceito` .

Exemplo 6 – Conversão de códigos numéricos em textos (HTTP status)

Contexto do problema:

Você tem um serviço que recebe códigos HTTP (200, 400, 404, 500) e precisa gerar uma explicação amigável para log, tela ou API de suporte.

Código completo:

java Copiar código

```java
public class Exemplo06ConversaoCodigos {

    public static void main(String[] args) {
        int statusCode = 404;

        String mensagem;
        switch (statusCode) {
            case 200:
                mensagem = "OK - Requisição bem sucedida.";
                break;
            case 201:
                mensagem = "CREATED - Recurso criado com sucesso.";
                break;
            case 400:
```

```
                mensagem = "BAD REQUEST - Requisição inválida.";
                break;
            case 401:
                mensagem = "UNAUTHORIZED - Não autorizado.";
                break;
            case 404:
                mensagem = "NOT FOUND - Recurso não encontrado.";
                break;
            case 500:
                mensagem = "INTERNAL SERVER ERROR - Erro interno no servidor.";
                break;
            default:
                mensagem = "Código HTTP desconhecido: " + statusCode;
        }

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

Casos de uso reais:

- Logs de API.
- Painéis administrativos.
- Mensagens amigáveis para usuários.

Vantagens:

- Centraliza a conversão de código → texto.
- Fica claro quais códigos são tratados.

Cuidados:

- Para muitos códigos, pode ficar gigante (neste caso, talvez um `Map` seja melhor).
- Atualizações frequentes em códigos podem exigir mexer no switch.

Passo a passo:

1. Define `statusCode`.
2. `switch` verifica valores conhecidos.
3. Atribui a mensagem correspondente ou default.
4. Imprime.

Exemplo 7 – Switch moderno com `->` e `yield` (Java 14+)

Contexto do problema:

Você tem planos de assinatura (`BASICO` , `PRO` , `EMPRESARIAL`) e quer calcular um valor de desconto retornando diretamente da expressão.

Código completo:

java Copiar código

```java
enum Plano {
    BASICO,
    PRO,
    EMPRESARIAL
}

public class Exemplo07SwitchExpression {

    public static void main(String[] args) {
        Plano plano = Plano.PRO;
        int numeroUsuarios = 15;

        double desconto = switch (plano) {
            case BASICO -> 0.0;
            case PRO -> {
                double base = 0.05; // 5%
                if (numeroUsuarios > 10) {
                    base += 0.02; // +2% se muitos usuários
                }
                yield base;
            }
            case EMPRESARIAL -> 0.10; // 10%
        };

        System.out.println("Plano: " + plano);
        System.out.println("Desconto aplicado: " + (desconto * 100) + "%");
    }
}
```

Casos de uso reais:

- Cálculo de desconto baseado em tipo de plano.
- Regras tributárias diferentes por operação.
- Seleção de políticas (frete, cobrança, limites).

Vantagens do switch expression:

- **Mais conciso**: você retorna diretamente um valor.
- **Sem fall-through** por padrão.
- **Exaustivo**: se esquecer um case de enum, o compilador reclama.

Cuidados:

- Necessário Java 14+ (idealmente 17 pra produção hoje em dia).
- Em blocos `{}` você precisa usar `yield`.

Passo a passo:

1. Define `enum Plano`.
2. Atribui `Plano.PRO` e número de usuários.
3. O `switch (plano)` retorna um `double` (atribui em `desconto`).
4. Para `PRO`, usa um bloco mais complexo e no final `yield base`.
5. Imprime o valor do desconto.

Exemplo 8 – Caso prático de mercado: roteamento de regra em API

Contexto do problema:
Imagine uma API de pagamentos que recebe um tipo de meio de pagamento: `"CARTAO"`, `"PIX"`, `"BOLETO"`. Cada um tem uma taxa e fluxo diferente. Você quer centralizar a escolha da regra.

Código completo (simples, mas com cara de "regra de negócio"):

java Copiar código

```java
public class Exemplo08SwitchAPIPagamento {

    public static void main(String[] args) {
        String meioPagamento = "PIX";
        double valor = 100.0;

        double taxa = calcularTaxa(meioPagamento, valor);
        double valorFinal = valor + taxa;

        System.out.println("Meio de pagamento: " + meioPagamento);
        System.out.println("Valor original: R$ " + valor);
        System.out.println("Taxa: R$ " + taxa);
        System.out.println("Valor final: R$ " + valorFinal);
    }
```

```
    private static double calcularTaxa(String meioPagamento, double valor) {
        return switch (meioPagamento) {
            case "CARTAO" -> valor * 0.03;  // 3%
            case "PIX"    -> valor * 0.005; // 0,5%
            case "BOLETO" -> 2.50;          // taxa fixa
            default       -> {
                System.out.println("Meio de pagamento desconhecido: " + meioPag
                yield 0.0;
            }
        };
    }
}
```

Casos de uso reais:

- APIs financeiras (pagamentos, transferências).
- Regras de frete por tipo de entrega.
- Pontos de fidelidade por tipo de operação.

Vantagens:

- `switch` expression deixa o código bem compacto.
- Fácil ver as taxas de cada meio de pagamento.

Cuidados:

- Em sistemas reais, isso provavelmente iria para um **Strategy** ou configuração externa.
- `String` literal pode ser trocada por `enum` para evitar erro de digitação.

Passo a passo:

1. `main` define meio de pagamento e valor.
2. Chama `calcularTaxa` .
3. O `switch` retorna diretamente a taxa:
    - `%` do valor para CARTAO e PIX.
    - Taxa fixa para BOLETO.
4. Caso desconhecido, loga e retorna 0.
5. Volta pro `main` , soma o valor com a taxa e imprime.

---

3. Boas práticas, armadilhas e comparações

3.1 Quando NÃO usar `switch`

Evite `switch` quando:

1. **Regras complexas** por combinação de condições:
   - `if (idade > 18 && renda > 5000 && possuiCartaoCredito) { ... }`
2. Você precisa de **intervalos** (ex.: de 0 a 100):
   - Use `if/else`, ou uma estrutura de classificação específica.
3. O número de casos é **muito grande** (dezenas/centenas):
   - Melhor usar uma estrutura de dados (`Map`, base de dados, config externa).
4. Lógica que muda com frequência:
   - `switch` exige alterar código e recompilar.
   - Talvez Strategy, banco ou arquivo de config seja melhor.

3.2 Switch vs. `Map` vs. Strategy Pattern

Switch

- **Bom para:**
  - Poucos casos, bem definidos.
  - Quando o mapeamento é estático e simples.
  - Você quer algo rápido de implementar.
- **Ruim para:**
  - Muitos casos.
  - Extensibilidade (adicionar novo tipo exige mexer na classe).

`Map`

Exemplo: converter código para mensagem usando `Map<Integer, String>`.

- **Vantagens:**
  - Fácil adicionar/remover entradas em tempo de execução.
  - Pode carregar de config externa (JSON, BD, etc).
- **Desvantagens:**
  - Menos "explícito" que o switch.
  - Erros de digitação podem não ser percebidos em compilação.

Uso típico:

java Copiar código

```java
Map<Integer, String> mensagens = new HashMap<>();
mensagens.put(200, "OK");
mensagens.put(404, "Não encontrado");
```

```
String msg = mensagens.getOrDefault(statusCode, "Desconhecido");
```

Strategy Pattern
Quando cada `case` do switch tem uma lógica complexa (chama serviço, faz cálculo, validação, etc), o switch vira um "Deus da lógica".

Strategy: você encapsula a lógica de cada caso em uma classe.

- **Vantagens:**
  - Código orientado a objetos, aberto para extensão.
  - Fácil testar cada estratégia isoladamente.
- **Desvantagens:**
  - Mais verboso; exige mais classes.
  - Overkill para casos simples.

Regra prática para ensinar:

> Se o seu `switch` está virando um **monstro de dezenas de linhas por** `case`, considere Strategy.

3.3 Armadilhas comuns com `switch`

1. **Esquecer o** `break` **(no switch tradicional):**

java Copiar código

```
switch (x) {
    case 1:
        System.out.println("1");
    case 2:
        System.out.println("2");
}
```

Se `x == 1`, imprime `1` e `2` (fall-through).

2. **Não tratar o** `default`:

- Sempre que possível, trate o caso inesperado.
- Em switch expression com `enum`, prefira cobrir todos os casos.

3. **Misturar contexto demais no** `switch`:

- Se cada `case` chama 10 métodos, tem `try/catch`, validações complexas etc., a classe fica difícil de manter.

4. **Usar `switch` onde `Map` seria mais simples:**
- Mapeamento "chave → valor" puro frequentemente é mais limpo via `Map`.

3.4 Regras de performance (sem exagero)

- Para tipos numéricos e `enum`, o `switch` geralmente é **muito eficiente** (jump table).
- `switch` costuma ser **mais rápido** que uma longa cadeia de `if/else if` em comparações simples, mas a diferença no dia a dia raramente é o gargalo.
- **Não otimize prematuramente:**
  - Priorize **legibilidade**.
  - Use profiler quando achar que virou gargalo.

Regras práticas para você falar em aula:

1. **Comece pela clareza:** se o switch deixa o código mais legível, use.
2. Se o mapeamento é gigante ou dinâmico → pense em `Map` ou BD.
3. Se a lógica varia muito e por tipo → pense em Strategy / polimorfismo.

Se você quiser, no próximo passo posso:

- Transformar isso em **slides** (títulos, bullets, exemplos resumidos).
- Montar um **roteiro de fala** para cada bloco da aula (o que comentar, onde fazer