

METODOLOGIA

- › Interprete o documento calmamente e com atenção.
- › Acompanhe a execução do exercício no seu computador.
- › Não hesite em consultar o formador para o esclarecimento de qualquer questão.
- › Não prossiga para o ponto seguinte sem ter compreendido totalmente o ponto anterior.
- › Caso seja necessário, execute várias vezes o exercício até ter compreendido totalmente o processo.

CONTEÚDO PROGRAMÁTICO

1. [Conceito](#)
2. [Galo](#)
 - 2.1. [Tabuleiro](#)
 - 2.2. [Fim do jogo](#)
3. [Escrita personalizada](#)
4. [Vantagens](#)

1. Conceito

Enumerado é um tipo de dados definido pelo programador que tem a sua gama de valores específica. Assim como um inteiro pode receber valores entre -2147483648 e 2147483647, um enumerado recebe valores de uma lista personalizada e definida no código.

Imagine que está a fazer um pequeno jogo e que precisa de guardar a cor de um semáforo. Como iria guardar a cor? Com o que vimos até este ponto a solução seria utilizar uma **String** para o nome da cor ou um inteiro em que cada número corresponde a uma cor. Embora possível, essa representação não seria a ideal, pois não o código fica menos evidente, como permite ao programador colocar valores que não são válidos. Com um enumerado a representação é simples e intuitiva, através da seguinte sintaxe:

```
public enum Semaforo {  
    Verde,  
    Laranja,  
    Vermelho  
}
```

A palavra reservada `enum` significa enumerado e cada valor é separado por uma vírgula.

Este enumerado passa a funcionar como um tipo de dados sobre o qual podemos criar variáveis e atribuir valores. Para atribuir a cor **Verde** a um semáforo utilizamos a seguinte sintaxe:

```
Semaforo corSemaforo1 = Semaforo.Verde;
```

Nesta linha de código, o valor é atribuído com `Semaforo.Verde`, em que **Semaforo** corresponde ao nome do enumerado e **Verde** a um dos valores possíveis.

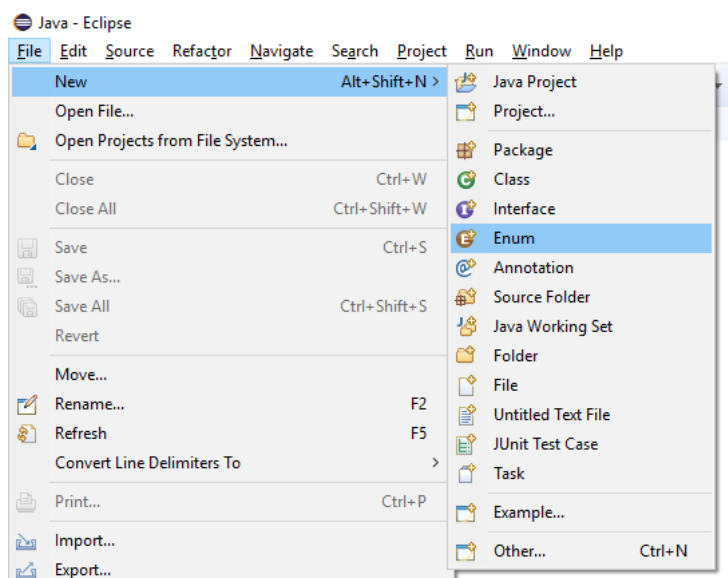
Vamos aplicar este conceito utilizando como exemplo um programa que trabalha com dias da semana. Primeiro é necessário construir o enumerado.

- › Clique em **File** 1, escolha a opção **New** 2 e **Enum** 3

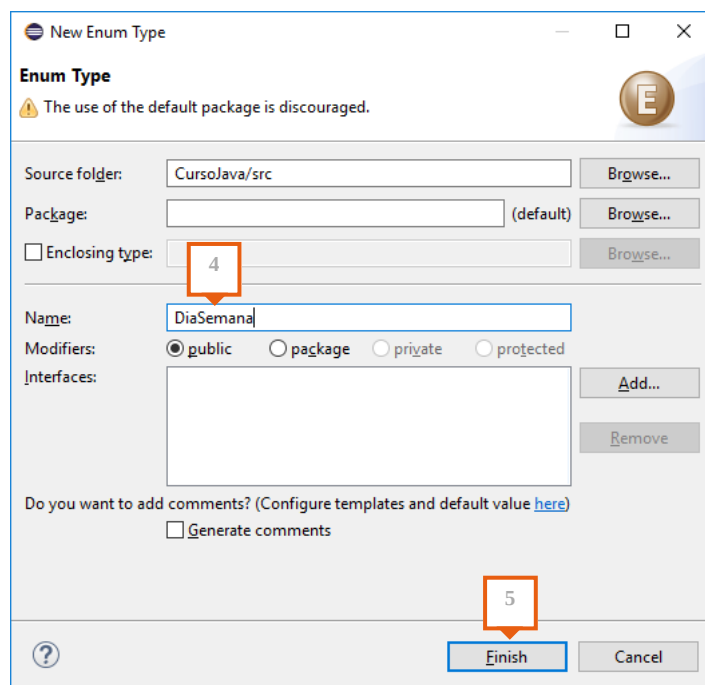
1

2

3



- › No quadro que surge, **defina** o nome **DiaSemana** 4 e clique em **Finish** 5



- › **Coloque** o código que se segue:

```
public enum DiaSemana {  
    Segunda,  
    Terça,  
    Quarta,  
    Quinta,  
    Sexta,  
    Sabado,  
    Domingo  
}
```

O enumerado já está criado, fornecendo as várias possibilidades para os dias da semana. Passemos à sua utilização numa classe com o método `main`.

- › **Crie** uma nova **classe** com o nome `DiaSemanaTeste` e **insira** o seguinte:

```
import java.util.Scanner;  
  
public class DiaSemanaTeste {  
    public static void main(String[] args) {  
        Scanner teclado = new Scanner(System.in);  
    }  
}
```

```

System.out.println("Escolha um dia da semana (Segunda/Terça/Quarta/Quinta/Sexta/Sábado/Domingo)");
String diaTexto = teclado.nextLine();

DiaSemana dia = DiaSemana.valueOf(diaTexto);

if (dia == DiaSemana.Domingo) {
    System.out.println("A Formabase está fechada no domingo");
}
else if (dia == DiaSemana.Sabado) {
    System.out.println("A Formabase está fechada sábado à tarde");
}
else {
    System.out.println("A Formabase funciona normalmente de segunda a sexta");
}
}
}

```

- › **Teste** o código **várias vezes** colocando valores diferentes para o dia da semana

Começamos por pedir ao utilizador o dia da semana pretendido e guardamos o valor obtido na variável `diaTexto`, como já foi visto anteriormente no curso.

```
String diaTexto = teclado.nextLine();
```

Na instrução que se segue criamos uma variável do tipo `DiaSemana`, que é um enumerado, convertendo o texto pedido ao utilizador para a opção equivalente:

```
DiaSemana dia = DiaSemana.valueOf(diaTexto);
```

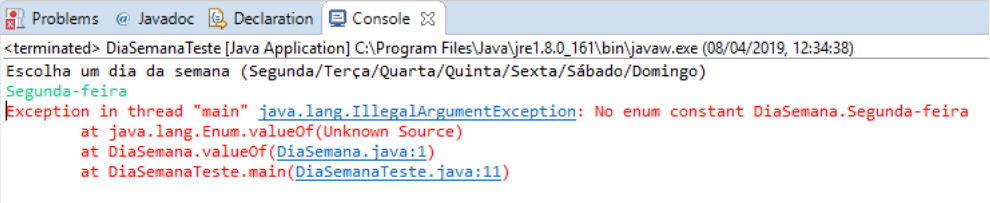
- 📌 O método `valueOf` de um enumerado obtém a opção que corresponde ao valor passado como parâmetro.

Após este passo a variável `dia` tem um dos 7 dias da semana possíveis e definidos por nós. Começamos por testar no `if` se corresponde ao valor `Domingo`:

```
if (dia == DiaSemana.Domingo) {
```

E prosseguimos com o teste para o `Sabado` e restantes dias da semana.

- › **Volte a executar** o programa e, como dia da semana, escreva `Segunda-feira`
- › **Observe** o resultado na consola



```

<terminated> DiaSemanaTeste [Java Application] C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe (08/04/2019, 12:34:38)
Escolha um dia da semana (Segunda/Terça/Quarta/Quinta/Sexta/Sábado/Domingo)
Segunda-feira
Exception in thread "main" java.lang.IllegalArgumentException: No enum constant DiaSemana.Segunda-feira
    at java.lang.Enum.valueOf(Unknown Source)
    at DiaSemana.valueOf(DiaSemana.java:1)
    at DiaSemanaTeste.main(DiaSemanaTeste.java:11)

```

Tal como quando lemos um inteiro, se o valor não fizer parte da gama válida de valores, obtemos uma exceção. Para contornarmos este problema necessitamos de capturar esta nova exceção do tipo `IllegalArgumentException`.

- › **Acrescente** na mesma classe as linhas realçadas:

```

public static void main(String[] args) {
    Scanner teclado = new Scanner(System.in);
    try {
        System.out.println("Escolha um dia da semana (Segunda/Terça/Quarta/Quinta/Sexta/Sábado/Domingo)");
        String diaTexto = teclado.nextLine();

        DiaSemana dia = DiaSemana.valueOf(diaTexto);

        if (dia == DiaSemana.Domingo) {
            System.out.println("A Formabase está fechada no domingo");
        }
        else if (dia == DiaSemana.Sábado) {
            System.out.println("A Formabase está fechada sábado à tarde");
        }
        else {
            System.out.println("A Formabase funciona normalmente de segunda a sexta");
        }
    }
}

```

```
        catch(IllegalArgumentException excecao) {  
            System.out.println("Dia da semana inválido");  
        }  
    }  
}
```

- › **Teste** e confirme que vê a mensagem de erro personalizado para um valor inválido

2. Galo

Recorda-se do jogo do Galo feito em módulos anteriores? Esse é um dos programas onde podemos aplicar enumerados, tornando o código mais claro e mais flexível.

2.1. Tabuleiro

A primeira alteração a fazer é no tabuleiro, que foi construído como um array bidimensional de caracteres:

```
static char[][] tabuleiro;
```

Como array de caracteres o tabuleiro pode ter outros valores para além dos que desejamos, mas através de um enumerado isso já não é possível.

A alteração vai refletir-se em todos os métodos do jogo e para mantermos a versão anterior do galo como referência, vamos criar uma nova versão para os enumerados. Para simplificar vamos também ignorar o tamanho e assumir um tabuleiro fixo de 3 por 3, de forma a dar mais ênfase na aplicação dos enumerados.

- › Crie um novo **package** chamado **galoenumerados**
- › No novo package **crie** um **enumerado** com o nome `Simbolo`
- › Para esse enumerado **escreva** o seguinte:

```
package galoenumerados;  
  
public enum Simbolo {  
    X,  
    O,  
    Vazio  
}
```

Com este novo ficheiro definimos as possibilidades de valores para cada uma das casas do tabuleiro. A inicialização dos valores do tabuleiro passa a ter a nova definição de vazio como valor inicial.

- › Ainda no mesmo package **crie** uma classe `Galo`
- › **Insira** o seguinte código:

```
package galoenumerados;  
  
import java.util.Scanner;  
  
public class Galo {  
    static Simbolo[][] tabuleiro;  
  
    public static void inicializacaoTabuleiro(){  
        //cria e define o tamanho do array  
        tabuleiro = new Simbolo[3][3];  
        for (int linha = 0; linha < 3; linha++){  
            for(int coluna = 0; coluna < 3; coluna++){  
                tabuleiro[linha][coluna] = Simbolo.Vazio;  
            }  
        }  
    }  
}
```

O `tabuleiro` passou a ser um array bidimensional de `Simbolo` em vez de `char`:

```
static Simbolo[][] tabuleiro;
```

A sua criação também utiliza o tipo `Simbolo`:

```
tabuleiro = new Simbolo[3][3];
```

E a atribuição dos valores iniciais foi feita com o símbolo `Vazio`:

```
tabuleiro[linha][coluna] = Simbolo.Vazio;
```

› **Acrescente** o método `mostraTabuleiro` ainda na mesma classe `Galo`:

```
public static void mostraTabuleiro(){
    for (int linha = 0; linha < 3; linha++){
        System.out.println("-----");
        System.out.print("| ");

        for(int coluna = 0; coluna < 3; coluna++){
            if (tabuleiro[linha][coluna] == Simbolo.Vazio) {
                //escrever Vazio como um espaço em branco
                System.out.print("  | ");
            }
            else {
                System.out.print(tabuleiro[linha][coluna] + " | ");
            }
        }
        System.out.println();
    }
    System.out.println("-----");
}
```

📖 Neste método que mostra no tabuleiro, apenas a inserção do valor de cada casa foi feita de forma diferente. Quando está vazia mostramos um espaço em branco, e quando tem `X` ou `O` mostramos diretamente esse valor. Isto evita que apareça o texto **Vazio** nas casas que estão vazias.

› **Acrescente** o método `fazerJogada` que se segue:

```
public static boolean fazerJogada(Simbolo simboloCorrente){
    Scanner teclado = new Scanner(System.in);

    System.out.println("Insira a linha onde quer jogar");
    int linha = teclado.nextInt();
    if(linha < 1 || linha > 3){
        //linha invalida
        return false;
    }

    System.out.println("Insira a coluna onde quer jogar");
    int coluna = teclado.nextInt();
    if(coluna < 1 || coluna > 3){
        //coluna invalida
        return false;
    }

    if(tabuleiro[linha - 1][coluna - 1] != Simbolo.Vazio){
        //quadrícula já preenchida
        return false;
    }
    //guardar a jogada que foi feita
    tabuleiro[linha - 1][coluna - 1] = simboloCorrente;
    return true; //devolver sucesso
}
```

No método `fazerJogada`, apenas a verificação da quadrícula já preenchida foi alterada:

```
if(tabuleiro[linha - 1][coluna - 1] != Simbolo.Vazio){
```

📖 Esta verificação agora utiliza o `Simbolo.Vazio` que foi definido no enumerado.

Para começarmos a testar o código precisamos do método principal, o `main`.

› **Adicione** à classe `Galo` o método `main` que se segue:

```
public static void main(String[] args) {
    inicializacaoTabuleiro();
    Simbolo simboloCorrente = Simbolo.X;

    while (true){
        boolean jogadaValida = fazerJogada(simboloCorrente);
        while(jogadaValida == false){
```

```

        System.out.println("Jogada invalida, jogue de novo");
        jogadaValida = fazerJogada(simboloCorrente);
    }

    if(simboloCorrente == Simbolo.X){
        simboloCorrente = Simbolo.O;
    }
    else {
        simboloCorrente = Simbolo.X;
    }

    mostraTabuleiro();
}
}

```

› **Teste** o jogo

- Repare que neste momento o jogo não termina, pois o `main` foi simplificado e a verificação para quando o jogo acaba ainda não existe.

A variável `simboloCorrente` passou a ser do tipo `Simbolo` que corresponde ao enumerado e é inicializada com o valor `X`:

```
Simbolo simboloCorrente = Simbolo.X;
```

O alternar de símbolo também passou a utilizar os valores definidos no enumerado:

```
if(simboloCorrente == Simbolo.X){
    simboloCorrente = Simbolo.O;
}
```

Neste momento o jogo não termina, pois o ciclo foi propositadamente definido para correr indefinidamente:

```
while (true){
```

Mais à frente iremos implementar a verificação de fim do jogo, que irá terminar este ciclo com a instrução `break`.

2.2. Fim do jogo

No módulo do jogo do Galo que fez anteriormente, implementou um método para detetar quando o jogo termina. Esse método tem a seguinte assinatura:

```
public static boolean jogoAcabado(){
```

O seu retorno é verdadeiro quando o jogo já terminou e falso caso contrário. Mas repare que o jogo tanto pode acabar com um empate ou uma vitória. Se quisermos mostrar uma mensagem personalizada para vitória ou empate no final do jogo torna-se mais complicado, pois temos de repetir a instrução várias vezes no método. Desta forma, podemos criar um enumerado para definir o estado de jogo corrente e retorná-lo no fim do método.

Primeiro começamos por criar o enumerado com os estados possíveis do jogo.

- › **Crie** no mesmo package um novo **enumerado** com o nome **EstadoJogo**

- › **Defina** o seu conteúdo com o **seguinte código**:

```
package galoenumerados;
public enum EstadoJogo {
    VitoriaX,
    VitoriaO,
    Empate,
    ADecorrer
}
```

- › **Acrescente** na classe `Galo` o método `obterEstado` que se segue:

```
public static EstadoJogo obterEstado(){
    //linhas
    for(int i = 0; i < 3; ++i){
        if(tabuleiro[i][0] == tabuleiro[i][1] && tabuleiro[i][1] == tabuleiro[i][2]){
            if (tabuleiro[i][0] == Simbolo.X) {
                return EstadoJogo.VitoriaX;
            }
            else if (tabuleiro[i][0] == Simbolo.O) {
                return EstadoJogo.VitoriaO;
            }
        }
    }
}
```

```

    }
}

//colunas
for(int i = 0; i < 3; ++i){
    if(tabuleiro[0][i] == tabuleiro[1][i] && tabuleiro[1][i] == tabuleiro[2][i]){
        if (tabuleiro[0][i] == Simbolo.X) {
            return EstadoJogo.VitoriaX;
        }
        else if (tabuleiro[0][i] == Simbolo.O) {
            return EstadoJogo.VitoriaO;
        }
    }
}

//diagonal esquerda
if(tabuleiro[0][0] == tabuleiro[1][1] && tabuleiro[1][1] == tabuleiro[2][2]){
    if (tabuleiro[0][0] == Simbolo.X) {
        return EstadoJogo.VitoriaX;
    }
    else if (tabuleiro[0][0] == Simbolo.O) {
        return EstadoJogo.VitoriaO;
    }
}

//diagonal direita
if(tabuleiro[0][2] == tabuleiro[1][1] && tabuleiro[1][1] == tabuleiro[2][0]){
    if (tabuleiro[0][2] == Simbolo.X) {
        return EstadoJogo.VitoriaX;
    }
    else if (tabuleiro[0][2] == Simbolo.O) {
        return EstadoJogo.VitoriaO;
    }
}

for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        if(tabuleiro[i][j] == Simbolo.Vazio){
            return EstadoJogo.ADecorrer;
        }
    }
}

return EstadoJogo.Empate;
}

```

📌 Note que o nome passou a ser `obterEstado`, pois é mais correto para o que agora faz.

O tipo de retorno deste novo método corresponde ao enumerado `EstadoJogo`:

```
public static EstadoJogo obterEstado(){
```

Isto significa que o valor a ser retornado no método tem de ser uma das 4 opções existentes no enumerado.

A verificação das linhas começa com um ciclo `for` que percorre cada linha do tabuleiro e verifica se as 3 casas dessa linha são iguais:

```

for(int i = 0; i < 3; ++i){
    if(tabuleiro[i][0] == tabuleiro[i][1] && tabuleiro[i][1] == tabuleiro[i][2]){

```

Se as 3 casas forem iguais, precisamos de saber se tem um `X` ou um `O` para indicar a vitória. Como todas têm o mesmo valor, basta pegar na primeira casa, que é a encontra em `tabuleiro[i][0]`:

```

    if (tabuleiro[i][0] == Simbolo.X) {
        return EstadoJogo.VitoriaX;
    }

```

E segue-se o teste equivalente para o símbolo `O`:

```

    else if (tabuleiro[i][0] == Simbolo.O) {
        return EstadoJogo.VitoriaO;
    }

```

A verificação das colunas e diagonais segue exatamente o mesmo princípio ajustando apenas as posições em que se acede no tabuleiro.

Mais para o fim temos dois ciclos que verificam a existência de casas vazias:

```
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        if(tabuleiro[i][j] == Simbolo.Vazio){
            return EstadoJogo.ADecorrer;
```

Até este ponto ainda não houve nenhuma vitória, caso contrário o método já teria retornado. Por esse motivo, se existir uma casa vazia significa que o jogo ainda terminou e, por isso, fazemos o retorno com o valor `EstadoJogo.ADecorrer`.

Se não houver nenhuma casa vazia no fim do método, estamos perante um empate e por isso finalizamos com:

```
return EstadoJogo.Empate;
```

Agora precisamos de dar uso a este método no `main`.

➤ **Acrescente** no `main` o código destacado:

```
public static void main(String[] args) {
    inicializacaoTabuleiro();
    Simbolo simboloCorrente = Simbolo.X;

    while (true){
        ...

        mostraTabuleiro();

        EstadoJogo estado = obterEstado();
        if (estado != EstadoJogo.ADecorrer) {
            switch(estado) {
                case Empate: System.out.println("Jogo empatado"); break;
                case VitoriaX: System.out.println("Ganhou o jogador X"); break;
                case Vitoria0: System.out.println("Ganhou o jogador 0"); break;
            }

            break; //terminar o while
        }
    }
}
```

📘 Os `...` simbolizam o código que já existe e que não foi alterado.

➤ **Teste** de novo o jogo e **confirme** que termina corretamente quer para vitórias ou empates

Antes do fim do `while` obtemos o estado do jogo chamando o novo método `obterEstado`:

```
EstadoJogo estado = obterEstado();
```

O estado é uma das 4 possibilidades que existem no enumerado. Como queremos saber se o jogo deve terminar, verificamos se o estado é diferente da opção `AD`

```
if (estado != EstadoJogo.ADecorrer) {
```

Se já terminou utilizamos um `switch` para mostrar a mensagem apropriada consoante a forma como o jogo terminou:

```
switch(estado) {
    case Empate: System.out.println("Jogo empatado"); break;
    case VitoriaX: System.out.println("Ganhou o jogador X"); break;
    case Vitoria0: System.out.println("Ganhou o jogador 0"); break;
}
```

Para que o programa termine precisamos de encerrar o ciclo `while`, através de um `break`:

```
break; //terminar o while
```

3. Escrita personalizada

Sempre que escrevemos o valor de uma opção do enumerado na consola, o resultado é o texto correspondente à própria opção. Isto nem sempre é o desejado, u
que as opções que definimos seguem as regras de convenções de variáveis não permitindo carateres especiais ou espaços. Podemos alterar este comportamento
utilizarmos um construtor e um método `toString` para o enumerado.

Vamos aplicar este princípio ao enumerado `Simbolo` do jogo do Galo desenvolvido no ponto anterior, para que cada símbolo tenha uma correspondência em texto
definida por nós.

➤ **Reescreva** o enumerado `Simbolo`:

```
package galoenumerados;

public enum Simbolo {
    X ("X"),
    O ("O"),
    Vazio (" ");

    private String valor;

    Simbolo(String val) {
        valor = val;
    }

    @Override
    public String toString() {
        return valor;
    }
}
```

Cada valor do enumerado tem agora a sua representação em texto à frente e dentro de parênteses:

```
X ("X"),
```

A última opção é terminada com um `;` (ponto e vírgula):

```
Vazio (" ");
```

Guardamos o texto de cada opção como `String`, que vai ficar associado ao campo `valor`:

```
private String valor;
```

Este campo é atribuído na construção de cada opção, como se fosse um construtor de uma classe:

```
Simbolo(String val) {
    valor = val;
}
```

A transformação da opção em texto é feita com o método `toString` que, neste caso, se limita a retornar o texto definido:

```
@Override
public String toString() {
    return valor;
}
```

📖 O `@Override` é uma notação facultativa que indica que este método está a ser sobrecarregado. Isto significa que o método foi herdado de uma cl
base, neste caso a classe `Object`, que é a base de todas as classes.

Após esta alteração no enumerado deixa de ser necessária a instrução `if...else` no método `mostraTabuleiro`:

```
if (tabuleiro[linha][coluna] == Simbolo.Vazio) {
    System.out.print("   | "); //escrever Vazio como um espaço em branco
}
else {
    System.out.print(tabuleiro[linha][coluna] + " | ");
}
```

📖 Agora podemos escrever diretamente o valor do enumerado.

➤ **Reescreva** o método `mostraTabuleiro` eliminando o `if...else` e **acrescentando** a linha realçada:

```
public static void mostraTabuleiro(){
    for (int linha = 0; linha < 3; linha++){
        System.out.println("-----");
        System.out.print("| ");

        for(int coluna = 0; coluna < 3; coluna++){
            System.out.print(tabuleiro[linha][coluna] + " | ");
        }
        System.out.print();
    }
    System.out.println("-----");
}
```

› **Volte a executar** o programa e confirme que vê o tabuleiro a ser mostrado corretamente

Dentro do segundo ciclo `for` os valores do enumerado são escritos diretamente, pois são mostrados com o texto definido no enumerado, não sendo necessária nenhuma condição.

4. Vantagens

Existem várias vantagens em usar enumerados, e as mais importantes são:

- **Legibilidade e clareza** – O código fica mais claro e legível, pois não temos números ou textos que correspondem a valores internos do programa. Estes valores internos nem sempre são evidentes a quem lê o código e não está dentro do contexto.
- **Robustez** – O código torna-se mais robusto, pois não é possível ao programador colocar um valor fora da gama definida.
- **Flexibilidade** – Torna-se mais fácil alterar qualquer um dos valores utilizados no enumerado através da opção Refactoring, que tem repercussões no resto do programa. Se o valor fosse um texto ou um número sem enumerados, o mesmo já não era tão fácil e, regra geral, implicava que o programador alterasse um pouco mais o código.