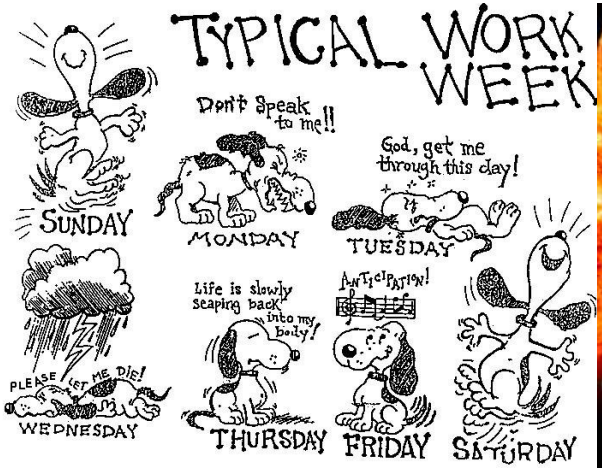
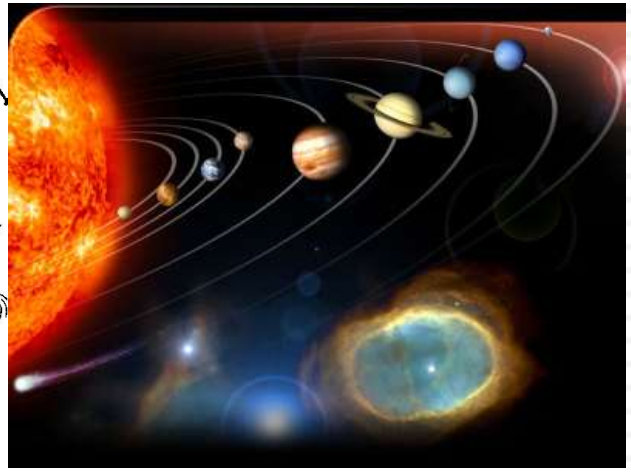

Programação Orientada Pelos Objectos
Enumerações

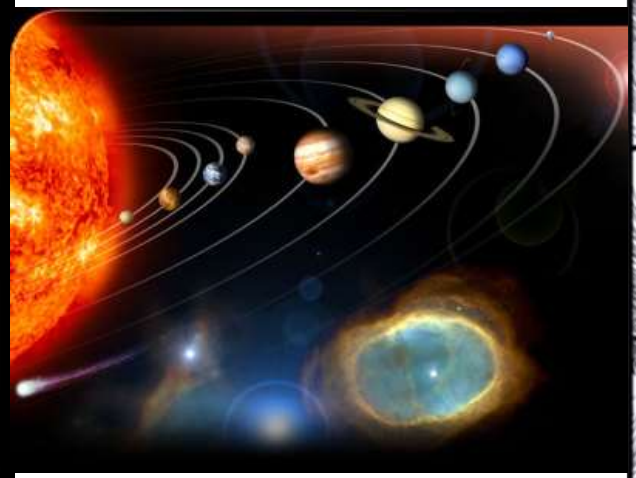
Tipos de dados Enumerados





Para que serve um tipo de dados enumerado?

- Construa uma aplicação que indica o peso de um corpo nos vários planetas do sistema solar



Fed up with how her diet is going, Charlene takes a more serious aim at her target weight.

Tipos enumerados

- Um tipo enumerado é um tipo cujos membros de dados são um conjunto fixo de constantes
- Exemplos típicos
  - Pontos cardeais
    - NORTH, SOUTH, EAST, WEST
  - Dias da semana
    - SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
  - Naipes de um baralho de cartas
    - CLUBS, HEARTS, DIAMONDS, SPADES
- Nota: representamos os campos do tipo enumerado com maiúsculas, dado que se tratam de **constantes** – mantemos assim a convenção habitual para as constantes

Definição de um tipo enumerado

- Define-se um tipo enumerado com a palavra reservada **`enum`**. Exemplos:
  - Pontos cardeais:

```
public enum CompassDirections {
   NORTH, SOUTH, EAST, WEST
}
```

  - Dias da semana:

```
public enum WeekDay {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

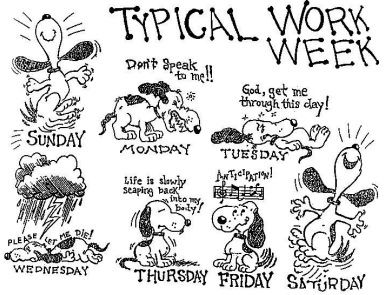
  - Naipes:

```
public enum CardSuit {
   CLUBS, HEARTS, DIAMONDS, SPADES
}
```

Quando usar tipos enumerados?

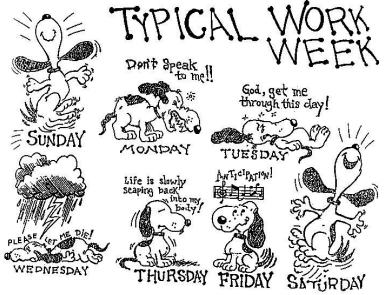
- Sempre que necessitamos de representar um conjunto fixo de constantes:
  - Tipos enumerados naturais
    - Pontos cardeais, dias da semana, meses do ano, ou planetas do sistema solar
  - Conjuntos de dados para os quais TODOS os valores possíveis são conhecidos em tempo de compilação
    - Escolhas de um menu, *flags* na linha de comando, etc.

Programamando a classe de teste




```
public class EnumTest {
  public static void dayDescription(WeekDay day) {
    System.out.print(day + " ");
    switch (day) {
      case MONDAY:
      case THURSDAY:
        System.out.println("Maravilha, aula teórica de POO. ;-)");
        break;
      case TUESDAY:
      case WEDNESDAY:
      case FRIDAY:
        System.out.println("Hoje é dia de Eclipse. Vampiros?!? :-[");
        break;
      case SATURDAY:
      case SUNDAY: System.out.println("Mais tempo para programar! :-)");
        break;
      default: System.out.println("Mas… há outros?");
        break;
    } // switch
  } // dayDescription
```

Programa principal

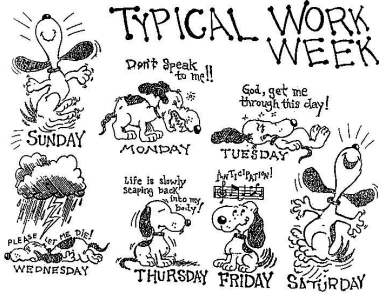



```
private static void repetitiveWeekDescription () {
   dayDescription(WeekDay.SUNDAY);
   dayDescription(WeekDay.MONDAY);
   dayDescription(WeekDay.TUESDAY);
   dayDescription(WeekDay.WEDNESDAY);
   dayDescription(WeekDay.THURSDAY);
   dayDescription(WeekDay.FRIDAY);
   dayDescription(WeekDay.SATURDAY);
}
```

Isto assim é muito repetitivo. Podemos fazer melhor!

```
SUNDAY Mais tempo para programar! :-)
MONDAY Maravilha, aula teórica de POO ;-)
TUESDAY Hoje é dia de Eclipse. Vampiros?!? :-[
WEDNESDAY Hoje é dia de Eclipse. Vampiros?!? :-[
THURSDAY Maravilha, aula teórica de POO ;-)
FRIDAY Hoje é dia de Eclipse. Vampiros?!? :-[
SATURDAY Mais tempo para programar! :-)
```

for-each




```
private static void cleverDayDescription() {
   for(WeekDay day: WeekDay.values())
      dayDescription(day);
}
```

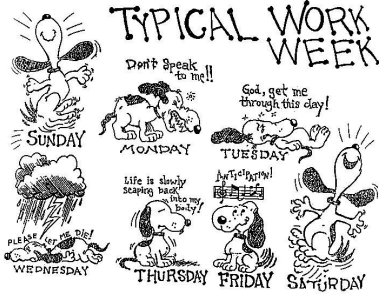
A operação `values()` devolve um vector de `WeekDay`. Esta operação é herdada da classe:

`java.lang.Enum`

O ciclo **`for-each`** visita todos os elementos da colecção devolvida por `WeekDay.values()`.

```
SUNDAY Mais tempo para programar! :-)
MONDAY Maravilha, aula teórica de POO ;-)
TUESDAY Hoje é dia de Eclipse. Vampiros?!? :-[
WEDNESDAY Hoje é dia de Eclipse. Vampiros?!? :-[
THURSDAY Maravilha, aula teórica de POO ;-)
FRIDAY Hoje é dia de Eclipse. Vampiros?!? :-[
SATURDAY Mais tempo para programar! :-)
```

for-each?




```
private static void cleverDayDescription() {
  for(WeekDay day: WeekDay.values())
    dayDescription(day);
}
```

- A construção `for-each` oferece uma forma compacta e menos propensa a erros de iterar todos os elementos de uma colecção
- A instrução destacada lê-se *"para cada valor* `day` *do tipo* `WeekDay` *na colecção* `WeekDay.values()`*"*
- Quando usamos? Sempre que possível ☺

O que é, afinal, um tipo enumerado em Java?

- Um tipo enumerado define uma classe que herda de `java.lang.Enum`
- A declaração do enumerado pode incluir métodos e outros campos
- Por ser sub-classe de `java.lang.Enum`, tem automaticamente algumas operações
  - `values()` devolve um vector com todos os valores do tipo enumerado, pela ordem com a qual foram declarados
  - Este método é normalmente usado em conjugação com o `for-each`

Consegue detectar o erro?


```
enum Suit {...};
enum Rank {...};


Array<Card> sortedDeck = new ArrayClass<>();
// BROKEN - throws NoSuchElementException!
for (Iterator<Suit> i = Suit.values().iterator(); i.hasNext();){

  for (Iterator<Rank> j = Rank.values().iterator(); j.hasNext();){
    sortedDeck.add(new Card(i.next(), j.next()));
  }
}
```

Problema resolvido!


```
enum Suit {...};
enum Rank {...};


Array<Card> sortedDeck = new ArrayClass<>();
// Fixed, though a bit ugly
for (Iterator<Suit> i = Suit.values().iterator(); i.hasNext();){
  Suit suit = (Suit) i.next();
  for (Iterator<Rank> j = Rank.values().iterator(); j.hasNext();){
    sortedDeck.add(new Card(suit, j.next()));
  }
}
```

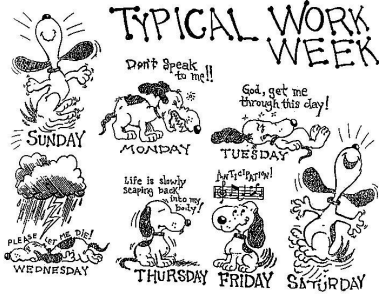
Problema evitado com `for-each`


```
enum Suit {...};
enum Rank {...};


Array<Card> sortedDeck = new ArrayClass<>();
// Fixed, much simpler, with for-each!
for (Suit suit: suits){
  for (Rank rank: ranks){
    sortedDeck.add(new Card(suit, rank));
  }
}
```

Quando **não usar** o `for-each`?

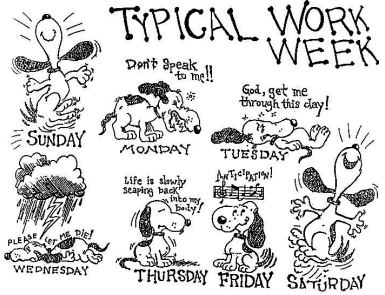
- O loop for-each "esconde" o iterador...
  - Se necessitar de aceder ao iterador, não consegue
    - Numa operação de filtragem que transforme a colecção, o for-each não é adequado
    - Numa operação em que tenha de transformar os elementos à medida que os visita, o for-each não é adequado
    - Se tiver de iterar várias colecções em paralelo, o for-each não é adequado

Voltemos ao **switch**
É muito fácil esquecer um **break**...



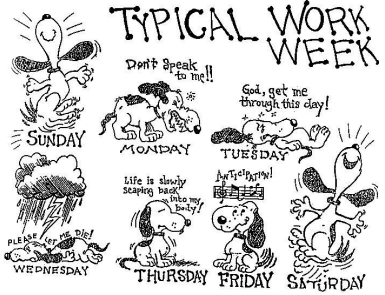

```
public class EnumTest {
  public static void dayDescription(WeekDay day) {
    System.out.print(day + " ");
    switch (day) {
      case MONDAY:
      case THURSDAY:
        System.out.println("Maravilha, aula teórica de POO. ;-)");
        break;
      case TUESDAY:
      case WEDNESDAY:
      case FRIDAY:
        System.out.println("Hoje é dia de Eclipse. Vampiros?!? :-[");
        break;
      case SATURDAY:
      case SUNDAY: System.out.println("Mais tempo para programar! :-)");
        break;
      default: System.out.println("Mas… há outros?");
        break;
    } // switch
  } // dayDescription
```

Podemos usar o novo **switch**!




```
public class EnumTest {
  public static void dayDescriptionNewSyntax(WeekDay day) {
    System.out.print(day + " ");
    switch (day) {
      case MONDAY, THURSDAY ->
        System.out.println("Maravilha, aula teórica de POO. ;-)");
      case TUESDAY, WEDNESDAY, FRIDAY ->
        System.out.println("Hoje é dia de Eclipse. Vampiros?!? :-[");
      case SATURDAY, SUNDAY ->
        System.out.println("Mais tempo para programar! :-)");
      default ->
        System.out.println("Mas… há outros?");
    } // switch
  } // dayDescription
```

Etiquetas `case L ->`

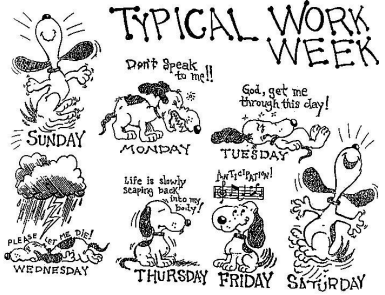



```
public class EnumTest {
  public static void dayDescriptionNewSyntax(WeekDay day) {
    System.out.print(day + " ");
    switch (day) {
      case MONDAY, THURSDAY ->
        System.out.println("Maravilha, aula teórica de POO. ;-)");
      case TUESDAY, WEDNESDAY, FRIDAY ->
        System.out.println("Hoje é dia de Eclipse. Vampiros?!? :-[");
      case SATURDAY, SUNDAY ->
        System.out.println("Mais tempo para programar! :-)");
      default ->
        System.out.println("Mas… há outros?");
    } // switch
  } // dayDescription
```

Pode conter etiquetas com o formato `case L ->` para eliminar a necessidade de usar a instrução **`break`** evitando assim eventuais esquecimentos!

Juntar etiquetas

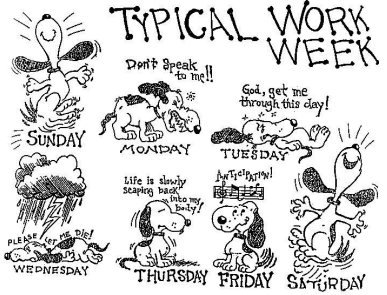
`case L, M, N ->`




```
public class EnumTest {
  public static void dayDescriptionNewSyntax(WeekDay day) {
    System.out.print(day + " ");
    switch (day) {
      case MONDAY, THURSDAY ->
        System.out.println("Maravilha, aula teórica de POO. ;-)");
      case TUESDAY, WEDNESDAY, FRIDAY ->
        System.out.println("Hoje é dia de Eclipse. Vampiros?!? :-[");
      case SATURDAY, SUNDAY ->
        System.out.println("Mais tempo para programar! :-)");
      default ->
        System.out.println("Mas… há outros?");
    } // switch
  } // dayDescription
```

Torna listas de casos com tratamento comum mais evidentes com o formato `case L, M, N ->`

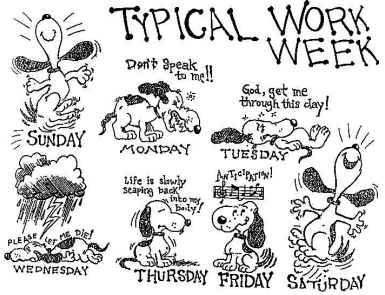
Definição exaustiva de casos




```
public class EnumTest {
  public static void dayDescriptionNewSyntax(WeekDay day) {
    System.out.print(day + " ");
    switch (day) {
      case MONDAY, THURSDAY ->
        System.out.println("Maravilha, aula teórica de POO. ;-)");
      case TUESDAY, WEDNESDAY, FRIDAY ->
        System.out.println("Hoje é dia de Eclipse. Vampiros?!? :-[");
      case SATURDAY, SUNDAY ->
        System.out.println("Mais tempo para programar! :-)");
    } // switch
  } // dayDescription
```

No caso do **`enum switch`**, o **`default`** pode ser omitido se definirmos o que acontece para todos os valores conhecidos (o compilador de Java acrescenta o **`default`**)

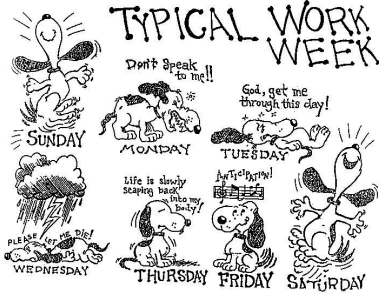
Imagine que queremos devolver o número de letras no dia da semana




```
public class EnumTest {
  public static int countLetters(WeekDay day) {
    return switch (day) {
      case MONDAY, FRIDAY, SUNDAY -> yield 6;
      case TUESDAY -> yield 7;
      case THURSDAY, SATURDAY -> yield 8;
      case WEDNESDAY -> yield 9;
    } // switch
  } // countLetters
```

Pode-se usar a instrução **`yield`** para o **`switch`** devolver um valor.

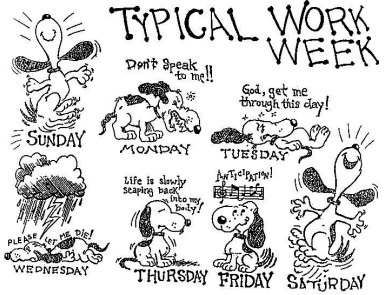
Imagine que queremos devolver o número de letras no dia da semana




```
public class EnumTest {
  public static int compactCountLetters(WeekDay day) {
    return switch (day) {
      case MONDAY, FRIDAY, SUNDAY -> 6;
      case TUESDAY -> 7;
      case THURSDAY, SATURDAY -> 8;
      case WEDNESDAY -> 9;
    } // switch
  } // countLetters
```

A instrução **`yield`** para o **`switch`** devolver um valor pode ser implícita, se os casos tiverem apenas uma instrução.

Imagine que queremos devolver o número de letras no dia da semana




```
public class EnumTest {
  public static int compactCountLetters(WeekDay day) {
    return switch (day) {
      case MONDAY, FRIDAY, SUNDAY -> {
        System.out.println("Seis"); yield 6;
      }
      case TUESDAY -> {
        System.out.println("Sete"); yield 7;
      }
      case THURSDAY, SATURDAY -> {
        System.out.println("Oito"); yield 8;
      }
      case WEDNESDAY -> {
        System.out.println("Nove"); yield 9;
      }
    } // switch
  } // compactCountLetters
```

Podemos usar um bloco de instruções para cada caso.


Um exemplo de peso do outro mundo!

Exemplo: Planetas


```
public enum Planet {

  private final double mass; // in kilograms
  private final double radius; // in meters

  private Planet(double mass, double radius) {
    this.mass = mass;
    this.radius = radius;
  }
```

Exemplo: Planetas


```
public enum Planet {
  // Planet (double mass, double radius)
  MERCURY (3.303e+23, 2.4397e6),
  VENUS   (4.869e+24, 6.0518e6),
  EARTH   (5.976e+24, 6.37814e6),
  MARS    (6.421e+23, 3.3972e6),
  JUPITER (1.9e+27,   7.1492e7),
  SATURN  (5.688e+26, 6.0268e7),
  URANUS  (8.686e+25, 2.5559e7),
  NEPTUNE (1.024e+26, 2.4746e7);

  private final double mass; // in kilograms
  private final double radius; // in meters

  private Planet(double mass, double radius) {
    this.mass = mass;
    this.radius = radius;
  }
```

Cada constante do enumerado `Planet` é declarada com valores para os parâmetros da massa e raio dos planetas. Estes valores são passados ao **`private`** construtor quando a constante é criada. As constantes têm de ser declaradas antes de quaisquer outros membros de dados, ou operações.

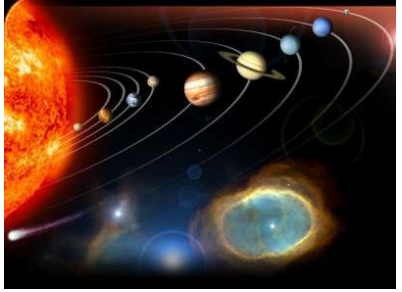
O construtor tem de ser privado, ou de visibilidade package. Não pode ser chamado de fora. Cria automaticamente as constantes declaradas no início do corpo da declaração do tipo enumerado.

Exemplo: Planetas


```
  private double mass() {
    return mass;
  }

  private double radius() {
    return radius;
  }

  // universal gravitational constant  (m3 kg-1 s-2)
  public static final double G = 6.67300E-11;

  public double surfaceGravity() {
    return G * mass() / (radius() * radius());
  }

  public double surfaceWeight(double otherMass) {
    return otherMass * surfaceGravity();
  }
}
```

O tipo enumerado contém outras operações e mesmo uma constante G.
Note que o valor de G, apesar de constante, **não é** um dos elementos do tipo enumerado.

Exemplo: Planetas




```
public class Main {
  private static final String WEIGHT_MSG = "Your weight on %s is %f\n";

  /**
   * Este programa permite calcular o peso de uma pessoa em cada um
   * dos planetas do sistema solar.
   * @param args
   */
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    double earthWeight = in.nextDouble();
    double mass = earthWeight/Planet.EARTH.surfaceGravity();
    for (Planet p: Planet.values())
        System.out.printf(WEIGHT_MSG, p, p.surfaceWeight(mass));
    in.close();
  }
}
```

Exemplo de interacção

- Exemplo de cálculo do peso de uma pessoa com 70kg (na terra), nos vários planetas do sistema solar

```
70
Your weight on MERCURY is 26.443033
Your weight on VENUS is 63.349937
Your weight on EARTH is 70.000000
Your weight on MARS is 26.511603
Your weight on JUPITER is 177.139027
Your weight on SATURN is 74.621088
Your weight on URANUS is 63.358904
Your weight on NEPTUNE is 79.682965
```

30

Comandos como Enumerados

Enumerado Command


```
public class Main {
    // Comandos do utilizador
    private enum Command {
        SAIR, REGISTA, CONSULTAPESSOA, AMIGOS, CONSULTAMIGOS,
        CONSULTAESTADO, NOVOESTADO, PESSOAS, UNKNOWN
    };
    ...

    private static Command getCommand(Scanner input) {
        try {
            String comm = input.nextLine().toUpperCase();
            return Command.valueOf(comm);
        } catch (IllegalArgumentException e) {
            // se o comando não for reconhecido
            return Command.UNKNOWN;
        }
    }
```

Há aqui um elemento novo que vamos ignorar por agora: este `try…catch`

main()


```
private static void main(String[] args) {
   SocialNetwork sn = new SocialNetworkClass();
   Scanner in = new Scanner(System.in);
   processCommands(in);
   in.close();
}
```

processCommands()


```
private static void processCommands(SocialNetwork sn,
                                    Scanner in) {
  Command command;
  do {
    command = getCommand(in);
    processCommand(sn, command, in);
  } while (!command.getValue().equals(Command.SAIR)) {
}
```

processCommand()


```
private static void processCommand(SocialNetwork sn,
                                   Command command,
                                   Scanner in) {
  switch (command) {
    case REGISTA: registerPerson(in, sn); break;
    case CONSULTAPESSOA : hasPerson(in, sn); break;
    case AMIGOS: setFriendship(in, sn); break;
    case CONSULTAMIGOS: numberOfFriends(in, sn); break;
    case CONSULTAESTADO: getStatus(in, sn); break;
    case NOVOESTADO: setStatus(in, sn); break;
    case PESSOAS: list(sn); break;
    case SAIR: quit(); break;
    default: break; // Não faz nada
  }
}
```

processCommand()


```
private static void processCommand(SocialNetwork sn,
                                   Command command,
                                   Scanner in) {
  switch (command) { // Com a nova sintaxe do switch
    case REGISTA -> registerPerson(in, sn);
    case CONSULTAPESSOA -> hasPerson(in, sn);
    case AMIGOS -> setFriendship(in, sn);
    case CONSULTAMIGOS -> numberOfFriends(in, sn);
    case CONSULTAESTADO -> getStatus(in, sn);
    case NOVOESTADO -> setStatus(in, sn);
    case PESSOAS -> list(sn);
    case SAIR -> quit();
    case UNKNOWN -> doNothing();
  }
}
```