ENUMERACIONES

Las **enumeraciones o enum** en un lenguaje de programación sirven para representar un grupo de constantes con un nombre. Veamos algunos ejemplos para comprender mejor.

1. Qué es un Enum.

En su forma más simple, **una enumeración es una lista de constantes con nombre que definen un nuevo tipo de datos**. Un objeto de un tipo de enumeración solo puede contener los valores definidos por la lista. Por lo tanto, una enumeración le brinda una manera de definir con precisión un nuevo tipo de datos que tiene **un número fijo de valores válidos.**
Desde una perspectiva de programación, las enumeraciones son útiles siempre que necesite definir un **conjunto de valores que represente una colección de elementos.** Por ejemplo, puede usar una enumeración para representar un conjunto de códigos de estado, como *éxito*, *espera*, *error* y *reintentos*, que indican el progreso de alguna acción. En el pasado, dichos valores se definían como **variables finales**, pero las enumeraciones ofrecen un enfoque más estructurado.

Los enum en Java se usan cuando conocemos todos los valores posibles en tiempo de compilación, como las opciones en un menú, los modos de redondeo, los indicadores de línea de comando, etc. No es necesario que el conjunto de constantes en un tipo de enumeración permanezca fijo para siempre.
En Java (desde 1.5), las enumeraciones se representan utilizando el tipo de datos **enum**. Las enumeraciones Java son más potentes que en C/C++. En Java, también podemos agregarle atributos, métodos y constructores. El objetivo principal de enum es definir nuestros propios tipos de datos (**tipos de datos enumerados**).

2. Declaración de enum en Java

La declaración de Enum puede hacerse fuera de **una clase**, o dentro de una clase (*class*), pero NO dentro de un método.

```
[acceso] enum NombreEnum{
        IDENTIFICADOR1[,IDENTIFICADOR2…];
        [atributos]
        [constructores]
        [métodos]
}
```

Los IDENTIFICADORES se denominan **constantes de enumeración** de acuerdo con la nomenclatura Java se recomienda nombrarlos en mayúculas. Cada uno se declara implícitamente como un miembro público (**public**) y estático (**static**) de *NombreEnum*. Además, el tipo de constantes de enumeración es el tipo de enumeración en el que se declaran las constantes, que es *NombreEnum*. Por lo tanto, en el lenguaje de Java, estas constantes se llaman **auto-tipado**.

3. Ejemplo básico con enum en Java

```
public enum Planeta{
   MERCURIO,VENUS,TIERRA,MARTE,JUPITER,SATURNO,URANO,NEPTUNO,PLUTON;
}
```

```
public class PruebaPlaneta{
   public static void main(String[] args){
   //Entorno:
          Planeta p1;
   //Algoritmo:
          p1 = Planeta.TIERRA;
          System.out.println(p1);
   }//Fin Programa
}
```

Salida:

```
TIERRA
```

3.1. Explicación del ejemplo.

```
Planeta p;
```

Como *p* es de tipo *Planeta*, los únicos valores que se le pueden asignar son los definidos por la enumeración. Por ejemplo, esto asigna a *p* el valor *VENUS*:

```
p = Planeta.VENUS;
```

Se pueden comparar dos constantes de enumeración utilizando el operador relacional **==**. Por ejemplo, esta declaración compara el valor en *p* con la constante *TIERRA*:

```
if(p == Planeta.TIERRA) // ...
```

Un valor de enumeración también se puede usar para controlar una sentencia switch. Por supuesto, todas las declaraciones de **case** deben usar constantes de la misma enumeración que la utilizada por la expresión de switch. Por ejemplo, este *switch* es perfectamente válido:

```
//Uso de enum para controlar una sentencia switch
switch(p){
   case VENUS:
        //
```

```
   case JUPITER:
        //
}//Fin Según Sea
```

CUIDADO en las sentencias *case*, los nombres de las constantes de enumeración se usan sin estar calificados por el nombre de tipo de enumeración. Es decir, se utiliza *VENUS*, no *Planeta.VENUS*. Esto se debe a que el tipo de enumeración en la expresión de *switch* ya ha especificado implícitamente el tipo de enumeración de las constantes de *case*.

- No es necesario calificar las constantes en las declaraciones de **case** con su nombre de tipo enum. De hecho, intentar hacerlo provocará un error de compilación.

Cuando se muestra una constante de enumeración, como en una instrucción *println()*, se genera su nombre. Por ejemplo, dada esta declaración:

```
System.out.println(Planeta.TIERRA);
```

se muestra el nombre TIERRA.

3.2. Otro Ejemplo

```
//Una enumeración de transporte
public enum Transporte{
   COCHE, CAMION, AVION, TREN, BARCO;
}
```

```
public class PruebaTransporte{
   public static void main(String[] args) {
   //Entorno:
        Transporte tp;
   //Algoritmo:
        tp=Transporte.AVION;
        System.out.println("Valor de tp: "+tp);
        System.out.println();
        tp=Transporte.TREN;
        //Comparación de 2 valores enum
        if (tp==Transporte.TREN){
           System.out.println("tp tiene el valor de TREN");
        }//Fin Si
        //enum para controlar sentencia switch
        switch(tp){
           case COCHE:
              System.out.println("Un coche lleva personas.");
              break;
           case CAMION:
              System.out.println("Un camión lleva carga.");
              break;
           case AVION:
              System.out.println("Un avión vuela.");
              break;
           case TREN:
              System.out.println("Un tren va por las vías.");
              break;
           case BARCO:
              System.out.println("Un barco navega en el agua.");
              break;
        }//Fin Según Sea
   }//Fin Programa
}
```

Salida:

```
Valor de tp: AVION

tp tiene el valor de TREN
```

```
Un tren va por las vías.
```

Antes de continuar, es necesario hacer un punto sobre estilo. Las constantes en *Transporte* usan mayúsculas. (Por lo tanto, se usa *COCHE*, *no coche*.) Sin embargo, **no se requiere el uso de mayúsculas**. En otras palabras, no existe una regla que requiera que las constantes de enumeración estén en mayúsculas.
Debido a que las enumeraciones a menudo reemplazan las variables finales, que tradicionalmente se han usado en mayúsculas, algunos programadores creen que las **constantes de enumeración en mayúsculas también son apropiadas**. Por supuesto, hay otros puntos de vista y estilos.

4. Puntos importantes de enum

- Cada enum es implementado internamente mediante el uso de *class*.

```
/* internamente enum Color se convierte en
class Color{
   public static final Color ROJO = new Color();
   public static final Color AZUL = new Color();
   public static final Color VERDE = new Color();
}*/
```

- Cada constante enum representa un objeto de tipo enum.
- El tipo enum se puede pasar como un argumento para *switch*.
- Cada constante enum siempre es implícitamente public static final. Entonces, como es static, podemos acceder utilizando el nombre del *enum*. Y, como es final, no podemos crear enumeraciones «*hijas*»

Aunque los ejemplos anteriores muestran la mecánica de crear y usar una enumeración, no muestran todas sus capacidades. A diferencia de la forma en que se implementan las enumeraciones en algunos otros lenguajes, **Java implementa enumeraciones como tipos de clases**. Aunque no crea una instancia de una enumeración usando new, actúa de forma muy similar a otras clases.
El hecho de que enum define una clase permite que la enumeración de Java tenga poderes que las enumeraciones en otros lenguajes no tienen. Por ejemplo, puede **darle constructores**, **agregar atributos** y **métodos de instancia**, e incluso **implementar interfaces**.

5. Métodos values(), ordinal() y valueOf()

Todas las enumeraciones tienen automáticamente dos métodos predefinidos: **values**() y **valueOf**(). Sus formas generales se muestran aquí:

```
public static tipo-enum[ ] values( )
public static tipo-enum valueOf(String str)
```

- Estos métodos están presentes dentro de **java.lang.Enum**.
- El método **values**() se puede usar para devolver todos los valores presentes dentro de enum.
- El orden es importante en las enumeraciones. Al usar el método **ordinal**(), se puede encontrar cada índice de la constante enum, al igual que el índice de matriz.
- El método **valueOf**() devuelve la constante enum del valor de cadena especificado, si existe.

Por ejemplo:

```
// Programa Java para demostrar el funcionamiento de values(),
// ordinal() y valueOf()
enum Color{
   ROJO, VERDE, AZUL;
}
```

```
public class Test{
   public static void main(String[] args){
   //Entorno:
      Color[] arr;
   //Algoritmo:
      // Llamando a values()
      arr = Color.values();
      // enum con bucle
      for (Color col : arr){
         // Llamando a ordinal() para encontrar el índice de color.
         System.out.println(col + " en el índice "
                            + col.ordinal());
      }//Fin Para
      // Usando valueOf(). Devuelve un objeto de
      // Color con la constante dada.
      // La segunda línea comentada causa la excepción
      // IllegalArgumentException
      System.out.println(Color.valueOf("ROJO"));
      // System.out.println(Color.valueOf("BLANCO"));
   }//Fin Programa
}
```

Salida:

```
ROJO en el índice 0
VERDE en el índice 1
AZUL en el índice 2
ROJO
```

6. enum: Constructores, métodos, variables de instancia

Es importante comprender que cada constante de enumeración es un objeto de su tipo de enumeración. Por lo tanto, **una enumeración puede definir constructores, agregar métodos y tener variables de instancia**.

Cuando define un constructor para una enumeración, se llama al constructor cuando se crea cada constante de enumeración. Cada constante de enumeración puede llamar a cualquier método definido por la enumeración. Cada constante de enumeración tiene su propia copia de cualquier variable de instancia definida por la enumeración.

- enum puede contener constructor y se ejecuta por separado para cada constante enum en el momento de la carga de la clase enum.
- No podemos crear objetos enum explícitamente y, por lo tanto, no podemos invocar el constructor enum directamente.

La siguiente versión de *Transporte* ilustra el uso de un constructor, una variable de instancia y un método. Da a cada tipo de transporte una velocidad típica:

```
//Uso de un constructor, una variable de instancia y un método.
enum Transporte{
   COCHE(60), CAMION(50), AVION(600), TREN(70), BARCO(20);
   private int velocidad; //velocidad típica de cada transporte

   //Añadir un onstructor
   Transporte(int s){
      velocidad=s;
   }
   //Añadir un método
   int getVelocidad(){
      return velocidad;
   }
}
```

```
class Enumerados {
   public static void main(String[] args) {
   //Entorno:
      Transporte tp;
   //Algoritmo:
```

```
      //Mostrar la velocidad de un avión
      System.out.println("La velocidad típica para un avión es: "+
      Transporte.AVION.getVelocidad()+ " km/h.");
      //Mostrar todas las velocidades y transportes
      System.out.println("Todas las velocidades de transporte: ");
      for (Transporte t:Transporte.values()){
         System.out.println(t+ ": velocidad típica es "+t.getVelocidad()+" km/h.");
      }//Fin Para
   }//Fin Programa
}
```

Salida:

```
La velocidad típica para un avión es: 600 millas por hora.

Todas las velocidades de transporte:
COCHE: velocidad típica es 60 km/h.
CAMION: velocidad típica es 50 km/h.
AVION: velocidad típica es 600 km/h.
TREN: velocidad típica es 70 km/h.
BARCO: velocidad típica es 20 km/h.
```

6.1. Explicación del Ejemplo

Esta versión de *Transporte* agrega tres cosas. La primera es el atributo de instancia ***velocidad***, que se usa para mantener la velocidad de cada tipo de transporte. El segundo es el constructor ***Transporte***, que pasa la *velocidad* de un transporte. El tercero es el método ***getVelocidad()***, que devuelve el valor de velocidad.
Cuando la variable ***tp*** se declara en ***main()***, el constructor de *Transporte* se llama una vez para cada constante que se especifica. Observe cómo se especifican los argumentos para el constructor, poniéndolos entre paréntesis, después de cada constante, como se muestra aquí:

```
COCHE(60), CAMION(50), AVION(600), TREN(70), BARCO(20);
```

Estos valores se pasan al parámetro de *Transporte()*, que luego asigna este valor a la *velocidad*. Hay algo más que notar sobre la lista de constantes de enumeración: **termina con un punto y coma**. Es decir, la última constante, *BARCO*, va seguida de un punto y coma. Cuando una enumeración contiene otros miembros, la lista de enumeración debe terminar en punto y coma.
Como cada constante de enumeración tiene su propia copia de velocidad, puede obtener la velocidad de un tipo de transporte especificado llamando a *getVelocidad()*. Por ejemplo, en *main()* la velocidad de un avión se obtiene mediante la siguiente llamada:

```
Transporte.AVION.getVelocidad()
```

La velocidad de cada transporte se obtiene al recorrer la enumeración mediante un bucle for. Como hay una copia de velocidad para cada constante de enumeración, el valor asociado con una constante es separado y distinto del valor asociado con otra constante. Este es un concepto potente, que está disponible solo cuando las enumeraciones se implementan como clases, como lo hace Java.
Aunque el ejemplo anterior contiene solo un constructor, una enumeración puede ofrecer dos o más formas de sobrecargas, al igual que cualquier otra clase.

7. Enum y Herencia

- Hay dos restricciones que se aplican a las enumeraciones. Primero, una enumeración no puede heredar otra clase. En segundo lugar, una enumeración no puede ser una superclase.

Esto significa que una enumeración no se puede extender. De lo contrario, enum actúa como cualquier otro tipo de clase. La clave es recordar que cada una de las constantes de enumeración es un objeto de la clase en la que está definida.

- Aunque no puede heredar una superclase al declarar una enumeración, todas las enumeraciones heredan automáticamente una: **java.lang.Enum**. Esta clase define varios métodos que están disponibles para el uso de todas las enumeraciones.
- Muy a menudo, no necesitará usar estos métodos, pero hay dos que puede emplear ocasionalmente: **ordinal()** y **compareTo()**.
- El método **toString()** se reemplaza en la clase java.lang.Enum, que devuelve el nombre de la constante enum.
- enum puede implementar muchas interfaces.