# 8 Tipos enumerados



## 1. Introducción

Un tipo enumerado (***enum type***) es un tipo cuyos únicos valores legales consisten en un conjunto fijo de constantes, como por ejemplo las estaciones del año o los palos en una baraja de cartas.

Antes de que los lenguajes nos permitieran definir tipos enumerados, éstos eran simulados declarando un grupo de constantes enteras, una para cada valor posible del tipo. Por ejemplo, para definir los tipos de manzanas y los tipos de naranjas podríamos hacer:

```
// Apple types.
public static final int APPLE_FUJI = 0;
public static final int APPLE_PIPPIN = 1;
public static final int APPLE_GRANNY_SMITH = 2;

// Orange types.
public static final int ORANGE_NAVEL = 0;
public static final int ORANGE_TEMPLE = 1;
```

Esta técnica, conocida como **el patrón de enumeración entera, tiene muchos defectos, por lo que no se recomienda su uso**.

Para empezar, al no definir realmente un tipo, el compilador no puede proporcionarnos seguridad real respecto al tipo. Por ejemplo, podemos pasar un tipo de naranja como argumento de un método que espera recibir un tipo de manzana, o almacenar un tipo de naranja en una variable que debe tener un tipo de manzana.

```
// Para el compilador esto es correcto.
int appleType = ORANGE_TEMPLE;
```

No solo eso, podemos comparar erróneamente tipos de manzanas y tipos de naranjas.

```
// Para el compilador esta comparación es válida.
int appleType = APPLE_PIPPIN;
int orangeType = ORANGE_TEMPLE;
if (appleType == orangeType) {
    // ...
}
```

Como Java no proporciona espacios de nombre para grupos de constantes enteras, debemos usar prefijos para evitar conflictos con los nombres de las constantes correspondientes a distintos grupos, como por ejemplo entre `ELEMENT_MERCURY` y `PLANET_MERCURY`.

Otro problema es que si se cambia el valor asociado a una constante, todas las clases que la usen seguirán compilando, pero su comportamiento puede llegar a ser incorrecto.

Finalmente, el patrón de enumeración entera tiene el inconveniente de que no hay una forma fácil de traducir las constantes enteras a cadenas imprimibles, es decir, la descripción textual del valor que desea representar.

En otras ocasiones se usa una variante del patrón anterior, pero usando el tipo `String` como tipo para las constantes en lugar de `int`. Por ejemplo:

```
// Apple types.
public static final String APPLE_FUJI = "FUJI";
public static final String APPLE_PIPPIN = "PIPPIN";
public static final String APPLE_GRANNY_SMITH = "SMITH";

// Orange types.
public static final String ORANGE_NAVEL = "NAVEL";
public static final String ORANGE_TEMPLE = "TEMPLE";
```

Esta variación se conoce como **patrón de enumeración con cadenas, y es aún menos deseable que la anterior**, porque aunque proporciona cadenas imprimibles para sus constantes, los desarrolladores pueden tener la tentación de usar en otras clases los valores de cadena directamente en vez de las constantes, por lo que un simple error tipográfico puede conllevar fallos en tiempo de ejecución. Además, la comparación de cadenas es mucho más costosa que la de enteros.

La alternativa adecuada a los patrones de enumeración anteriores es que el lenguaje de programación nos permita **definir tipos enumerados**, es decir, que nos permita definir un nuevo tipo indicando cuáles son los valores legales para dicho tipo y con qué nombre queremos referirnos a dichos valores.

En algunos lenguajes de programación, estos tipos enumerados son internamente tipos enteros. Sin embargo, en Java cuando definimos un tipo enumerado, estamos definiendo una clase completa, lo que nos proporciona una mayor funcionalidad a la de otros lenguajes de programación. En la versión 5 de Java, se incorporaron al lenguaje los tipos de datos enumerados.

Así, siguiendo el ejemplo anterior, definiríamos las siguientes clases `enum`:

```
public enum Apple  {
    FUJI, PIPPIN, GRANNY_SMITH
}

public enum Orange {
    NAVEL, TEMPLE
}
```

Los *enums* presentan numerosas ventajas frente al uso de los patrones de enumeración entera y de enumeración con cadenas. La más importante es que la clase *enum* corresponderá a un tipo, por lo que el compilador va a ser capaz de proporcionar comprobación de tipos en tiempo de compilación. Por ejemplo, si se declara que un parámetro es del tipo `Apple`, el compilador garantiza que cualquier referencia de objeto no nulo pasada al parámetro debe ser alguno de las instancias de `Apple` válidas que hemos definido en el *enum*. En caso contrario, se producirá un error de compilación. Ocurre exactamente lo mismo al asignar una expresión de un tipo de enumeración a una variable de un tipo *enum*, o al usar el operador == para comparar valores de diferentes tipos *enum*.

Además, los tipos *enum* definen su propio espacio de nombres, por lo que no hay ningún problema en usar constantes con el mismo nombre en distintos *enums*.

Otra ventaja es que si cambiamos el orden de las constantes en el *enum*, esto no afecta al código de los clientes del *enum*, dado que las constantes no son compiladas en el código cliente, a diferencia de como ocurría en los patrones anteriores.

## 2. Definición

Los enumerados se definen con la palabra reservada **enum**, el nombre del enumerado y luego el conjunto de las constantes, que por las convenciones del lenguaje se escriben en mayúscula. Las constantes de la enumeración son *public* y *static* de forma implícita.Ejemplo:

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

En el *Package Explorer* del *Eclipse*, hay que colocarse en el paquete donde se quiera crear el enum. Luego se pulsa el botón derecho del ratón y *New → Enum*.

Por otra parte, ¿dónde debemos definir un *enum*? Si va a ser usado desde varios lugares, lo lógico es definir el *enum* como una clase en su propio fichero. Si su uso está restringido a una única clase cliente, es más razonable definir el *enum* como una clase interna miembro de dicha clase.

Una vez definido el *enum*, se pueden crear variables de ese tipo:

```
DayOfWeek day;  //day es una variable del tipo de enumeración DayOfWeek
```

Como *day* es de tipo *DayOfWeek*, los únicos valores que se le pueden asignar son los definidos por la enumeración:

```
day = DayOfWeek.MONDAY; //Se asigna a day el valor MONDAY
```

El compilador nos da un error si intentamos asignar a una variable *enum* un tipo que no le corresponde:

```
DayOfWeek day = Apple.FUJI;//Error de compilación: falta de correspondencia
entre los tipos: no se puede convertir de Apple a DayOfWeek
```

## 3. Constructores

Los *enum* se implementan internamente como clases. El *enum DayOfWeek* internamente se convierte en:

```
class DayOfWeek {
    public static final DayOfWeek MONDAY = new DayOfWeek();
    public static final DayOfWeek TUESDAY = new DayOfWeek();
    public static final DayOfWeek WEDNESDAY = new DayOfWeek();
    ...
}
```

Los *enum* de Java son clases que exportan una instancia (objeto) para cada constante de enumeración a través de un campo final estático público. Estas instancias son creadas automáticamente al hacer referencia al *enum* en nuestro código, es decir, el constructor se ejecuta para cada constante en el momento de la carga de la clase *enum*. No podemos crear objetos *enum* explícitamente y, por lo tanto, no podemos invocar al constructor directamente:

```
DayOfWeek day = new DayOfWeek(); //Error de compilación: no se puede
instanciar el tipo DayOfWeek
```

Los únicos modificadores de acceso permitidos para los constructores de los *enum* son *friendly* y *private*. Los modificadores de acceso *protected* y *public* dan un error de compilación.

Como vemos, los `enum` de Java son una generación del patrón de diseño *singleton* permitiendo una determinada serie de instancias, cada una de las cuales es accesible a través de una constante. Visto de forma inversa, podríamos decir que un *singleton* no es más que un `enum` con una sola instancia.

## 4. Comparaciones

Los *enum* se pueden comparar utilizando el operador relacional **==**:

```
if(day == DayOfWeek.MONDAY) {//Si day contiene MONDAY, se ejecuta el bloque if
    ...
}
```

```
DayOfWeek day = DayOfWeek.MONDAY;
System.out.println(day == DayOfWeek.MONDAY ? true : false);//Imprime true
```

El compilador nos da un error si intentamos comparar valores de diferentes tipos *enum*:

```
DayOfWeek day = DayOfWeek.MONDAY;
Apple apple = Apple.FUJI;
if(day == apple) {//Error de compilación: tipos de operandos incompatibles
    ...
}
```

## 5. Enums en switch

También podemos utilizar **switch** para comprobar en los **case** los distintos valores del *enum*:

```
switch (day) {
case MONDAY:
    System.out.println("Lunes");
    break;
case THURSDAY:
    System.out.println("Martes");
    break;
...
}
```

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:

```
switch (day) {
case DayOfWeek.MONDAY://Error de compilación
    System.out.println("Lunes");
    break;
case THURSDAY:
    System.out.println("Martes");
    break;
...
}
```

## 6. Atributos

Los *enums* no dejan de ser clases, por lo que podemos declararles atributos, que deben ser **final**, ya que los enums son inmutables por naturaleza. Aunque estos atributos pueden ser *public*, se recomienda definirlos como **private** y definir los *getters* correspondientes si es necesario acceder a los atributos desde fuera del *enum*.

Para establecer el valor de estos atributos, debemos definir en la clase *enum* el constructor adecuado y al establecer las instancias pasaremos como argumento los valores deseados que se pasarán al constructor. En el ejemplo, cuando definimos *PLUS*, le ponemos entre paréntesis la cadena "+". Cuando se cree el objeto para *PLUS*, se ejecuta el constructor `private Operation(String symbol)` y en *symbol* se pasa "+":

```
package tema8_TiposEnumerados;

public enum Operation {

    PLUS("+"), MINUS("-"), TIMES("*"), DIVIDE("/");//El ; es necesario cuando
se definen atributos, constructores, etc..

    private final String symbol;

    private Operation(String symbol) {
        this.symbol = symbol;
    }

    public String getSymbol() {
        return symbol;
    }

}
```

```
package tema8_TiposEnumerados;

public class EnumAttributes {

    public void show() {

        Operation operation1 = Operation.PLUS;
        Operation operation2 = Operation.MINUS;
        System.out.printf("El atributo de PLUS es una cadena con el valor %s",
operation1.getSymbol());
        System.out.printf("\nEl atributo de MINUS es una cadena con el valor
%s", operation2.getSymbol());

    }
```

```
    public static void main(String[] args) {

        new EnumAttributes().show();

    }

}
```

Salida por consola:

```
El atributo de PLUS es una cadena con el valor +
El atributo de MINUS es una cadena con el valor -
```

# 7. Métodos

Una enumeración no puede heredar de otra clase ni puede actuar como superclase de otra clase, pero por defecto, hereda de la clase `java.lang.Enum<E>` y por tanto tiene una serie de métodos heredados, como por ejemplo los métodos `toString()` y `ordinal()`.

Además, todas las enumeraciones tienen automáticamente dos métodos estáticos predefinidos: `values()` y `valueOf(String str)`.

## 7.1 Método toString

Las clases *enum* proporcionan de serie implementaciones de alta calidad de todos los métodos de la clase `Object` (la clase `Enum` hereda de `Object`), como el método `public String toString()`, que retornará el nombre de la constante asociada a la instancia correspondiente. Por ejemplo `Apple.FUJI.toString()` retorna la cadena `FUJI`, aunque si no estamos contentos con dicha implementación, podemos sobrescribir nosotros dicho método. Además, las clases *enum* implementan de serie las interfaces `Comparable` y `Serializable`.

```
package tema8_TiposEnumerados;

public class EnumToString {

    public void show() {

        Operation operation1 = Operation.PLUS;
        Operation operation2 = Operation.MINUS;
        System.out.println(operation1.toString());//PLUS
        System.out.println(operation2.toString());//MINUS
        System.out.println(Operation.PLUS.toString());//PLUS
        System.out.println(Operation.MINUS.toString());//MINUS
    }

    public static void main(String[] args) {

        new EnumToString().show();

    }

}
```

De hecho, por defecto se llama al método `toString` cuando se imprime un *enum* o una variable de tipo *enum*:

```
package tema8_TiposEnumerados;
```

```
public class EnumPrinting {

    public void show() {

        Operation operation1 = Operation.PLUS;
        Operation operation2 = Operation.MINUS;
        System.out.println(operation1);//PLUS
        System.out.println(operation2);//MINUS
        System.out.println(Operation.PLUS);//PLUS
        System.out.println(Operation.MINUS);//MINUS

    }

    public static void main(String[] args) {

        new EnumPrinting().show();

    }

}
```

## 7.2 Método valueOf

`public static E valueOf(String name)` es un método declarado implícitamente que devuelve la instancia del *enum* que posee asociado el nombre de constante recibida.

```
package tema8_TiposEnumerados;

public class EnumValueOf1 {

    public void show() {

        Operation operation;
        operation = Operation.valueOf("PLUS");//operation se asigna con la
instancia correspondiente a la constante de enumeración PLUS
        System.out.printf("La variable operation es de tipo enum %s y su
símbolo es %s", operation,
                operation.getSymbol());

    }

    public static void main(String[] args) {

        new EnumValueOf1().show();

    }

}
```

Salida por consola:

```
La variable operation es de tipo enum PLUS y su símbolo es +
```

Si el argumento cadena que recibe el método *valueOf* no corresponde a ninguna constante de enumeración, el método lanza la excepción `IllegalArgumentException`:

```
package tema8_TiposEnumerados;

public class EnumValueOf2 {

    public void show() {
```

```
        Operation operation;
        operation = Operation.valueOf("PLUS1");//PLUS1 no se corresponde con
ninguna constante de enumeración
        System.out.printf("La variable operation es de tipo enum %s y su
símbolo es %s", operation,
                operation.getSymbol());

    }

    public static void main(String[] args) {

        new EnumValueOf2().show();

    }

}
```

Salida por consola:

```
Exception in thread "main" java.lang.IllegalArgumentException: No enum
constant tema8_TiposEnumerados.Operation.PLUS1
    at java.base/java.lang.Enum.valueOf(Enum.java:264)
    at tema8_TiposEnumerados.Operation.valueOf(Operation.java:1)
    at tema8_TiposEnumerados.EnumValueOf2.show(EnumValueOf2.java:8)
    at tema8_TiposEnumerados.EnumValueOf2.main(EnumValueOf2.java:16)
```

## 7.3 Método values

`public static E[] values()` es un método declarado implícitamente que devuelve un array con todas las instancias de la clase *enum* en el orden en que fueron declarados.

```
package tema8_TiposEnumerados;

public class EnumValues {

    public void show() {

        // Uso de values() en un for-each
        for (Operation operation : Operation.values()) {
            System.out.println(operation);
        }

    }

    public static void main(String[] args) {

        new EnumValues().show();

    }

}
```

Salida por consola:

```
PLUS
MINUS
TIMES
DIVIDE
```

## 7.4 Método ordinal

Una enumeración hereda de la clase `java.lang.Enum<E>` y por tanto tiene una serie de métodos heredados, como por ejemplo `ordinal()`.

`public final int ordinal()` devuelve un entero que indica la posición de la constante dentro de la enumeración. La primera constante tiene el valor 0.

```
package tema8_TiposEnumerados;

public class EnumOrdinal {

    public void show() {

        System.out.println(Operation.PLUS.ordinal());//0
        System.out.println(Operation.MINUS.ordinal());//1
        System.out.println(Operation.TIMES.ordinal());//2
        System.out.println(Operation.DIVIDE.ordinal());//3

    }

    public static void main(String[] args) {

        new EnumOrdinal().show();

    }

}
```

El problema de usar el método `ordinal()` es que el funcionamiento del código cliente depende completamente del orden en el que se definen las constantes en el *enum*, lo que puede resultar problemático para el mantenimiento del código si en un futuro se cambia dicho orden. En realidad, este método está diseñado para ser utilizado por estructuras de datos sofisticadas basadas en enumeraciones como `EnumSet` y `EnumMap`. (Se explican en el *Tema 11. Colecciones*).

Por lo tanto se recomienda no hacer uso del método `ordinal()`, sino definir un atributo y asignar un valor para dicho atributo en cada instancia:

```
package tema8_TiposEnumerados.ordinal;

public enum Operation {

    PLUS(1, "+"), MINUS(2, "-"), TIMES(3, "*"), DIVIDE(4, "/");

    private final int optionNumber;
    private final String symbol;

    private Operation(int optionNumber, String symbol) {
        this.optionNumber = optionNumber;
        this.symbol = symbol;
    }

    public String getSymbol() {
        return symbol;
    }

    public int getOptionNumber() {
        return optionNumber;
    }

}
```

```java
package tema8_TiposEnumerados.ordinal;

public class EnumAttributeInsteadOfOrdinal {

    public void show() {

        System.out.println(Operation.PLUS.getOptionNumber());//1
        System.out.println(Operation.MINUS.getOptionNumber());//2
        System.out.println(Operation.TIMES.getOptionNumber());//3
        System.out.println(Operation.DIVIDE.getOptionNumber());//4

    }

    public static void main(String[] args) {

        new EnumAttributeInsteadOfOrdinal().show();

    }

}
```

## 8. Métodos abstractos

En algunas ocasiones es necesario asociar un comportamiento ligeramente diferente en cada instancia del *enum*. En estos casos, debemos declarar un método abstracto en el *enum* e incluir la implementación concreta de dicho método abstracto en la definición de cada instancia. Estas implementaciones reciben el nombre de *constant-specific methods*. Por ejemplo:

```java
package tema8_TiposEnumerados.metodosAbstractos;

public enum Operation {

    PLUS("+") {
        @Override
        public double apply(double x, double y) {
            return x + y;
        }
    },
    MINUS("-") {
        @Override
        public double apply(double x, double y) {
            return x - y;
        }
    },
    TIMES("*") {
        @Override
        public double apply(double x, double y) {
            return x * y;
        }
    },
    DIVIDE("/") {
        @Override
        public double apply(double x, double y) {
            return x / y;
        }
    };

    private final String symbol;

    private Operation(String symbol) {
        this.symbol = symbol;
    }
```

```
    public String getSymbol() {
        return symbol;
    }

    public abstract double apply(double x, double y);
}
```

```
package tema8_TiposEnumerados.metodosAbstractos;

public class EnumAbstractMethods {

    public void show() {

        for (Operation operation : Operation.values()) {
            System.out.printf("%-8s  %.2f %s %.2f = %.2f\n", operation + ":",
5f, operation.getSymbol(), 3f,
                    operation.apply(5, 3));
        }

    }

    public static void main(String[] args) {

        new EnumAbstractMethods().show();

    }

}
```

Salida por consola:

```
PLUS:     5,00 + 3,00 = 8,00
MINUS:    5,00 - 3,00 = 2,00
TIMES:    5,00 * 3,00 = 15,00
DIVIDE:   5,00 / 3,00 = 1,67
```

## 9. Implementación de interfaces

Como ya hemos comentado, un *enum* no puede heredar de otro *enum*. Pero hay ocasiones donde tiene sentido que otros programadores puedan "extender" un *enum* que hayamos proporcionado en nuestra API. Por ejemplo cuando el *enum* corresponda a códigos de operación sobre una determinada máquina, como por ejemplo el *enum* `Operation`, sería recomendable que quien use nuestra API puede "añadir" nuevas operaciones. Para estos casos podemos simular la herencia entre *enums* definiendo una interfaz con el método de la operación y hacer que nuestro *enum* implemente dicha interfaz. Por ejemplo:

```
package tema8_TiposEnumerados.implementacionInterfaces;

public interface Operation {

    double apply(double x, double y);

    String getSymbol();

}
```

```
package tema8_TiposEnumerados.implementacionInterfaces;
```

```java
public enum BasicOperation implements Operation {

    PLUS("+") {
        @Override
        public double apply(double x, double y) {
            return x + y;
        }
    },
    MINUS("-") {
        @Override
        public double apply(double x, double y) {
            return x - y;
        }
    },
    TIMES("*") {
        @Override
        public double apply(double x, double y) {
            return x * y;
        }
    },
    DIVIDE("/") {
        @Override
        public double apply(double x, double y) {
            return x / y;
        }
    };

    private final String symbol;

    private BasicOperation(String symbol) {
        this.symbol = symbol;
    }

    @Override
    public String getSymbol() {
        return symbol;
    }

}
```

De esta manera, si otro equipo de desarrolladores quiere "extender" nuestro *enum* simplemente debe crear otro *enum* distinto que implemente la misma interfaz:

```java
package tema8_TiposEnumerados.implementacionInterfaces;

public enum ExtendedOperation implements Operation {

    EXP("^") {
        @Override
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }
    },
    REMAINDER("%") {
        @Override
        public double apply(double x, double y) {
            return x % y;
        }
    };

    private final String symbol;

    private ExtendedOperation(String symbol) {
        this.symbol = symbol;
```

```
    }

    @Override
    public String getSymbol() {
        return symbol;
    }

}
```

De esta manera, el código cliente que quiera trabajar con una operación, tendrá que definir la variable correspondiente de tipo interfaz:

```
Operation operation = ExtendedOperation.EXP;
```

El problema principal de esta técnica es que dado que no se trata de herencia real, no podemos reutilizar código entre los *enums*.

Veamos un ejemplo de uso de ambos *enums*:

```
package tema8_TiposEnumerados.implementacionInterfaces;

public class InterfacesImplementation {

    public void show() {

        for (Operation operation : BasicOperation.values()) {
            show(operation);
        }
        for (Operation operation : ExtendedOperation.values()) {
            show(operation);
        }

    }

    public void show(Operation operation) {

        System.out.printf("%-10s  %.2f %s %.2f = %.2f\n", operation + ":", 5f,
operation.getSymbol(), 3f,
                operation.apply(5, 3));

    }

    public static void main(String[] args) {

        new InterfacesImplementation().show();

    }

}
```

Salida por consola:

```
PLUS:       5,00 + 3,00 = 8,00
MINUS:      5,00 - 3,00 = 2,00
TIMES:      5,00 * 3,00 = 15,00
DIVIDE:     5,00 / 3,00 = 1,67
EXP:        5,00 ^ 3,00 = 125,00
REMAINDER:  5,00 % 3,00 = 2,00
```