Java

Enum
Interfaces
Generics

Enum

En su forma más simple se usan como los enum de C: como "contenedores de constantes"

```
static final int SUCCESS = 1;
static final int GOOD = 2;
static final int REGULAR = 3;
static final int BAD = 4;
```

```
public enum Rating {
   SUCCESS,
   GOOD,
   REGULAR,
   BAD
}
```

SUCCESS, GOOD, REGULAR y BAD son las únicas instancias de Rating

Enum

Como cualquier clase puede tener métodos de clase

```
public enum Rating {

  SUCCESS,  GOOD,  REGULAR,  BAD;

  public static String toString(Rating rating) {
     if ( rating == null)
        return "";
     switch ( rating) {
        case SUCCESS: return "Success";
        case GOOD: return "Good";
        case REGULAR: return "Regular";
        case BAD: return "Bad";
     }
     return "";
  }
```

Enum

Como cualquier clase puede tener métodos de clase

```
  public static Integer intValue(Rating rating) {
      if ( rating == null)
         return null;
      switch ( rating) {
         case SUCCESS: return 10;
         case GOOD: return 7;
         case REGULAR: return 5;
         case BAD: return 2;
         default: return 0;
      }
   }

}
```

Enum

Como cualquier clase puede tener métodos de instancia

```
public enum Rating {

  SUCCESS,  GOOD,  REGULAR,  BAD;

  @Override
  public String toString() {
    switch ( this) {
       case SUCCESS: return "Success";
       case GOOD: return "Good";
       case REGULAR: return "Regular";
       case BAD: return "Bad";
    }
    return "";
}
```

Enum

Cada uno de los "valores" del enum son las posibles instancia de la clase, y por lo tanto puede tener variables de instancia

```
public enum Rating{

  SUCCESS("Success", 10),
  GOOD("Good", 7),
  REGULAR("Regular", 5),
  BAD("Bad", 2);

  private final String description;
  private final int value;


  Rating (String description, int value) {
      this.description = description;
      this.value = value;
  }
```

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

  public Integer intValue() {
      return value;
  }

}
```

Ejercicio

Escribir una clase o método que permita realizar una operación matemática binaria representada por tres Strings, los dos primeros los operandos y el tercero el operador.

```
"12.5"  "34.0"  "+"          -> 36.5
"12.5"  "4.0"  "-"          -> 8.5
"4"  "12.0"  "-"            -> -8.0
"0.5"  "2"  "*"             -> 1.0
"a" "0.5"  "+"              -> alguna excepción
```

Para obtener el valor usamos `Double.valueOf(`*`string`*`)`

Ejercicio

Opción 1: creamos una clase con un método que reciba los tres strings, y en base al operador retorne el resultado

```
public class Evaluator {
   public Double evaluate(String op1, String op2, String op) {

        double n1 = Double.valueOf(op1);
        double n2 = Double.valueOf(op2);

        switch (op) {
            case "+": return n1 + n2;
            case "-": return n1 - n2;
            case "*": return n1 * n2;
            case "/": return n1 / n2;
        }
        return null;
    }
}
```

Ejercicio

Opción 2: creamos distintas clases para cada operador

```
public abstract class Operator  {
   public abstract double apply(double operand1, double operand2);
}
```

```
public class Sum extends Operator {

   @Override
   public double apply(double operand1, double operand2) {
       return operand1 + operand2;
   }
   @Override
   public String toString() {
       return "+";
   }
}
```

Ejercicio

Opción 2: creamos distintas clases para cada operador

```
public class Evaluator {
   public Double Evaluate(String op1, String op2, String op) {

        double n1 = Double.valueOf(op1);
        double n2 = Double.valueOf(op2);

        switch (op) {
            case "+": return new Sum().apply(n1, n2);
            case "-": return new Sub().apply(n1, n2);
            case "*": return new Mult().apply(n1, n2);
            case "/": return new Div().apply(n1, n2);
        }
        return null;
    }
}
```

Ejercicio

Opción 2 mejorada: usamos el patrón Singleton, y un método que determine la clase a usar

```
public abstract class Operator  {
   public abstract double apply(double operand1, double operand2);
}
```

```
public class Sum extends Operator {
   private static final Sum instance = new Sum();

   public static Sum getInstance() {
       return instance;
   }

   private Sum() { }

   @Override
   public double apply(double operand1, double operand2) {
       return operand1 + operand2;
   }
   @Override
   public String toString() {
       return "+";
   }
}
```

Ejercicio

Opción 2 mejorada: usamos el patrón Singleton, y un método que determine la clase a usar

```
public class Evaluator {

   public Double evaluate(String op1, String op2, String op) {

        Operator oper = getOperator(op);
        if (op == null)
            return null;

        return oper.apply(Double.valueOf(op1), Double.valueOf(op2));

   }

...
```

Ejercicio

Opción 2 mejorada: usamos el patrón Singleton, y un método que determine la clase a usar

```
    public Operator getOperator(String token) {
        if (token.length() != 1) {
            return null;
        }
        switch (token) {
            case "+":
                return Sum.getInstance();
            case "-":
                return Subtraction.getInstance();
            case "*":
                return Multiplication.getInstance();
            default:
                return null;
        }
    }
}
```

Ejercicio

Opción 3: en vez de Singleton, usar Enum

```
public enum Operations {
    ADD{
        public double apply(double a, double b) {
            return a + b;
        }},
    SUBTRACT {
        public double apply(double a, double b) {
            return a - b;
        }},
    MULTIPLY {
        public double apply(double a, double b) {
            return a * b;
        }};
    public abstract double apply(double a, double b);
}
```

Ejercicio

Opción 3: en vez de Singleton, usar Enum

```
public enum Operations {
   private final String symbol;
   ADD("+") {
       public double apply(double a, double b) {
           return a + b;
       }},
   SUBTRACT("-") {
       public double apply(double a, double b) {
           return a - b;
       }},
   MULTIPLY("*") {
       public double apply(double a, double b) {
           return a * b;
       }};

   Operations(String symbol
       this.symbol = symbol
```

```
    public abstract double apply(double a, double b);

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

Interfaces

- Una interface es una referencia a un tipo en Java
- Es una colección de métodos abstractos y constantes
- A partir de JDK 1.8 los métodos pueden tener un comportamiento por defecto
- Una clase puede implementar una o más interfaces.
- Un método puede especificar que retorne o recibe una interface. En tiempo de ejecución retornará o recibirá una instancia de una clase que implemente dicha interface
- Definición similar a una clase (package, .java, .class, etc.)
- Diferencias con una clase
  - No se pueden instanciar
  - No se puede especificar un constructor
  - No contiene variables de instancia
  - Una clase no extiende sino que implementa una interface
  - Una interface puede extender múltiples interfaces

Interfaces

Ejemplo: existen métodos (por ejemplo insertar ordenado en una colección, ordenar un vector, etc.) que necesiten comparar dos elementos. Ese método puede exigir que las clases que le pasen implementen una interface con un método de comparación

```
public interface Comparable {
   public int compareTo(Object other);
}
```

Esta interface no hace buen uso del lenguaje. Para ver mejores ejemplos antes tenemos que explicar Generics

```
public interface Comparable<T> {
   int compareTo(<T> other);
}
```

Generics

Permiten la parametrización de tipos de datos (clases e interfaces)

Permiten escribir código reusable por objetos de distinto tipo

```
class Pair<T> {
   private T first;
   private T second;

   public Pair(T first, T second) {
       this.first = first;
       this.second = second;
   }
   ...
}
```

```
class Pair<K, V> {
   private K key;
   private V value;

   public Pair(K key, V value) {
       this.key = key;
       this.value = value;
   }
   ...
}
```

Generics

Ejemplo: un despachante de aduanas maneja operaciones marítimas, terrestres y aéreas. Para cada una tiene tarifarios (con los costos) y en base a los tarifarios emiten cotizaciones a sus clientes

```
public class TariffAir {...}
public class TariffGround {...}
public class TariffOcean {...}

public class QuotationAir {
   private TariffAir tariff;
   public QuotationAir(Client client, Date date, TariffAir tariff) {
            ...
   }
```

¿Y si necesito ver las cotizaciones de un cliente?

¿No habrá métodos comunes a las distintas opciones?

Generics

```
public abstract class Tariff {...}
public class TariffAir extends Tariff {...}
public class TariffGround extends Tariff {...}
public class TariffOcean extends Tariff {...}

public abstract class Quotation {
   private Tariff tariff;

   public Quotation(Client client, Date date, Tariff tariff) {
            ...
   }

   public Tariff getTariff() { ... }
}
```

¿Cómo verificar que una cotización aérea referencia a un tarifario aéreo?

Generics

```
public abstract class Tariff {...}
public class TariffAir extends Tariff {...}
public class TariffGround extends Tariff {...}
public class TariffOcean extends Tariff {...}
```

```
public abstract class Quotation<T extends Tariff> {
   private T tariff;
   public Quotation(Client client, Date date, T tariff) {
            ...
   }
}
```

```
public class QuotationAir extends Quotation<Tariffair> {
   public QuotationAir(Client client, Date date, TairffAir tariff) {
            ...
   }
}
```

Arrays genéricos

```
Pair<String, String>[] v = new Pair[100];

v[0] = new Pair<>("100","Cien");

v[1] = new Pair(100.10, "Cien con 10");

Object[] v2 = v;

v2[2] = new Pair(100, "One hundred");

v = v2;
```

Generics: definición de una lista

```
public class LinkedList<E> {
        public E get(int index) { ... }
        public void add(E elem) { ... }
        ...
}
```

```
LinkedList<Integer> l = new LinkedList<Integer>();
```

```
public class ArrayList<E> {
        public E get(int index) { ... }
        public void add(E elem) { ... }
        ...
}
```

```
public class DoubleLinkedList<E> {
        public E get(int index) { ... }
        public void add(E elem) { ... }
        ...
}
```

¿Y si quiero hacer un método que muestre todos o algunos elementos de una lista cualquiera?

Generics

```
public interface List<E> {
    boolean contains(E elem);
    E get(int index);
    void add(E elem);

    default E getFirst() {
        return get(0);
    }
}
```

```
List<Integer> l = new ArrayList<Integer>();
```

```
List<Integer> l = new ArrayList<>();
```

```
Double sum(List<Double> aList)
{
    ...
}
```

```
void show(List aList) {
    ...
}
```

```
void show(List<?> aList) {
    ...
}
```

Generics: definición de una lista

```
public class LinkedList<E> implements List<E>{
            public E get(int index) { ... }
            public void add(E elem) { ... }
            ...
}
```

```
public class ArrayList<E> implements List<E> {
            public E get(int index) { ... }
            public void add(E elem) { ... }
            ...
}
```

```
public class DoubleLinkedList<E> implements List<E>{
            public E get(int index) { ... }
            public void add(E elem) { ... }
            ...
}
```

Comparator

Comparable exigía que la instancia sepa compararse contra otra instancia.
Comparator recibe dos instancias y las compara

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);

    ...
}
```

Ejemplo: una colección ordenada

Opción 1: La lista estará ordenada en base al "orden natural"

```
class SortedList<T extends Comparable<T>>
      implements List<T>{


}
```

Los elementos a insertar deben implementar Comparable

```
class SortedList<T extends Comparable<? super T>>
implements List<T>{


}
```

Los elementos a insertar o una clase ancestro debe implementar Comparable

Ejemplo: una colección ordenada

Opción 2: La lista estará ordenada por un criterio específico

```
class SortedList<T> implements List<T>{
   private Comparator<? super T> cmp;
   SortedList(Comparator<? super T> cmp) {
     this.cmp = cmp;
   }
}
```

Generics: métodos parametrizados

Se quiere pasar los elementos de un array a una lista

```
static void fromArrayToList(Object[] a, List<?> l) {
   for (Object o : a) {
       l.add(o); // error de compilación
   }
}
```

```
static void fromArrayToList(Object[] a, List<Object> l) {

```

```
static <T> void fromArrayToList(T[] a, List<T> l) {
   for (T o : a) {
       l.add(o);
   }
}
```

Generics vs *wildcards*

```
interface Collection<E> {
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c);
}
```

```
interface Collection<E> {
   <T> boolean containsAll(Collection<T> c);
   <T extends E> boolean addAll(Collection<T> c);
}
```

Generics

Si el valor de retorno no depende del tipo parametrizado, se debe usar *wildcard.*
Usaremos métodos parametrizados si el tipo de valor de retorno es genérico.

Ejemplo: el despachante de aduana persiste todos los cambios hechos a una cotización.
Se necesita un método que retorne el estado a una revisión determinada

```
public <Q extends Quotation> Q getVersion(Q quotation, int revision)
{
        Q old;
        ...
        return old;
}
```

Interfaces

Un uso que se le puede dar a las interfaces es como "marcador": una interfaz que no contiene métodos.

Las clases que las implementan no implementan comportamiento sino que son "marcadas" para que otros métodos tomen alguna acción al respecto

```
interface SecureResource {
}
```

NO hacer esto. Desde Java 1.5 existe una mejor forma de "marcar" o "anotar" una clase.

```
void showPage(Page page) {
    if ( page instanceof SecureResource ){
        ...
    }
}
```

Interfaces funcionales

Una interfaz con un único método abstracto puede implementarse tanto en forma "tradicional" como en forma funcional.

```
package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface Function<T, R> {
      R apply(T t);
            ...
}
```

```
void applyToArray(Double v[], Function<Double, Double> fn)
{
   for( Double x : v) {
       x = fn.apply(x);
   }
}
```

No está cambiando los elementos del vector.

Interfaces funcionales

Una interfaz con un único método abstracto puede implementarse tanto en forma "tradicional" como en forma funcional.

```
package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface Function<T, R> {
      R apply(T t);
            ...
}
```

```
void applyToArray(Double v[], Function<Double, Double> fn)
{
      for( int i= 0; i < v.length; i++) {
                        v[i] = fn.apply(v[i]);
   }
}
```

Para usar el método `applyToArray` podemos pasarle como segundo parámetro:

- Una instancia de una clase que implemente `Function<Double, Double>`
- Crear una clase anónima
- Una expresión lambda (notación funcional)

Interfaces funcionales

```
class Sin implements Function<Double , Double> {

   @Override
   public Double apply(Double number) {
       return Math.sin(number);
   }
}
```

```
Double v[] = new Double[] {1.0, 2.0, 1.5, 0.5};

applyToArray(v, new Sin());

applyToArray(v, new Function<Double, Double>() {
   @Override
   public Double apply(Double aDouble) {
       return Math.sin(aDouble);
   }
});

applyToArray(v, x -> Math.sin(x));
```

Function

Una interfaz funcional puede además contener constantes y métodos default

```
@FunctionalInterface
public interface Function<T, R> {

   ...

   default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
       Objects.requireNonNull(before);
       return (V v) -> apply(before.apply(v));
   }

   default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
       Objects.requireNonNull(after);
       return (T t) -> after.apply(apply(t));
   }

   static <T> Function<T, T> identity() {
       return t -> t;
   }
}
```